

L'Intelligence Artificielle pour les développeurs

Concepts et implémentations en C#

Virginie MATHIVET

Téléchargement
www.editions-eni.fr

eni Editions



L'Intelligence Artificielle pour les développeurs

Concepts et implémentations en C#

ISBN : 978-2-7460-9215-0
Imprimé en France

Editions ENI

ZAC du Moulin Neuf
Rue Benjamin Franklin
44800 St HERBLAIN
Tél. 02.51.80.15.15
Fax 02.51.80.15.16

e-mail : editions@ediENI.com
http://www.editions-eni.com

Auteur : Virginie MATHIVET
Collection **DataPro** dirigée par Joëlle MUSSET

Les exemples à télécharger sont disponibles à l'adresse suivante :
http://www.editions-eni.fr
Saisissez la référence ENI de l'ouvrage **DPINT** dans la zone de recherche et validez. Cliquez sur le titre du livre puis sur le bouton de téléchargement.

Avant-propos

Introduction

Chapitre 1
Systèmes experts

Chapitre 2
Logique floue

Chapitre 3
Recherche de chemins

Chapitre 4
Algorithmes génétiques

Chapitre 5
Métaheuristiques d'optimisation

Chapitre 6
Systèmes multi-agents

Chapitre 7
Réseaux de neurones

Bibliographie

Sitographie

Annexe

Avant-propos

1. Objectifs du livre

L'intelligence artificielle ou I.A. est un domaine qui passionne les amateurs de science-fiction. Cependant, dans notre monde actuel, de nombreux développeurs n'utilisent pas les techniques associées, par manque de connaissances de celles-ci.

Ce livre présente donc les principales techniques d'intelligence artificielle, en commençant par les concepts principaux à comprendre, puis en donnant des exemples de code en C#.

2. Public et pré-requis

Ce livre s'adresse à tous ceux qui souhaitent découvrir l'intelligence artificielle. Chaque chapitre détaille une technique.

Aucun pré-requis en mathématiques n'est requis, les formules et équations ayant été limitées au strict minimum. En effet, ce livre est surtout orienté sur les concepts et les principes sous-jacents aux différentes techniques.

La deuxième partie de chaque chapitre présente des exemples de code en C#. Une connaissance du langage, au moins basique, est nécessaire. Ce livre est donc surtout destiné aux développeurs, en particulier :

- Les **étudiants en école post-bac** qui souhaitent mieux comprendre l'intelligence artificielle, et étudier des exemples de code.
- Les **développeurs** qui doivent utiliser une technologie particulière et qui souhaitent trouver une explication des principes ainsi que des portions de code réutilisables et/ou adaptables.
- Les **passionnés** qui souhaitent découvrir l'intelligence artificielle et coder des programmes l'utilisant.
- Tout **curieux** qui s'intéresse à ce domaine.

3. Structure du livre

Ce livre commence par une **introduction**, permettant d'expliquer ce qu'est l'intelligence en général et l'intelligence artificielle en particulier. Les principaux domaines de celle-ci sont aussi présentés.

Le livre contient ensuite sept chapitres. Chacun d'eux porte sur une technique ou un ensemble de techniques. À l'intérieur de ceux-ci, on trouve tout d'abord l'explication des principes et des concepts. Ensuite suivent des exemples d'application de ces algorithmes et un code commenté et expliqué.

Le lecteur curieux ou voulant en apprendre sur plusieurs techniques pourra lire le livre dans l'ordre. Sinon, le lecteur cherchant des informations sur une technique particulière pourra aller directement au chapitre la concernant, ceux-ci étant indépendants.

Le premier chapitre présente les **systèmes experts**, permettant d'appliquer des règles pour faire du diagnostic ou de l'aide à un professionnel.

Le deuxième chapitre s'intéresse à la **logique floue**, permettant d'avoir des contrôleurs au comportement plus souple et plus proche du fonctionnement d'un humain.

Le troisième chapitre porte sur la **recherche de chemins**, en particulier les chemins les plus courts, sur une carte ou dans un graphe. Plusieurs algorithmes sont alors présentés (recherche en profondeur, en largeur, Bellman-Ford, Dijskstra et A*).

Le quatrième chapitre concerne les **algorithmes génétiques**. Ceux-ci s'inspirent de l'évolution biologique pour faire évoluer des solutions potentielles à des problèmes, jusqu'à trouver de bonnes solutions après plusieurs générations.

Le cinquième chapitre présente plusieurs **métaheuristiques d'optimisation**. Cinq algorithmes (algorithme glouton, par descente de gradient, recherche tabou, recuit simulé et optimisation par essaims particulaires) permettant d'améliorer des solutions sont présentés et comparés.

Le sixième chapitre s'intéresse aux **systèmes multi-agents**, dans lesquels plusieurs individus artificiels aux comportements simples vont, ensemble, résoudre des problèmes complexes.

Le dernier chapitre concerne les **réseaux de neurones**, permettant d'apprendre à résoudre des problèmes dont on ne connaît pas forcément le fonctionnement sous-jacent.

À la fin de ce livre se trouvent :

- Une **bibliographie**, permettant d'aller plus loin sur ces différentes techniques.
- Une **sitographie**, présentant des articles concernant l'utilisation réelle de ces algorithmes.
- Une **annexe** permettant d'installer et d'utiliser SWI Prolog, qui vient en complément du C# dans le chapitre sur les systèmes experts.
- Un **index**.

4. Code en téléchargement

Le code des différents chapitres est proposé en téléchargement sur le site de l'éditeur. Une **solution**, faite avec Visual Studio 2013, est disponible par chapitre.

Pour ouvrir ces solutions, il est possible de télécharger une version gratuite de **Visual Studio** : la version Express. Elle est disponible sur le site de Microsoft à l'adresse suivante :
http://www.visualstudio.com/fr-fr/products/visual-studio-express-vs.aspx

Dans chacune de ces solutions, la majorité du code (au moins toutes les classes au cœur des algorithmes) a été codée dans des **bibliothèques de classes portables** (ou PCL). De cette façon, ces classes peuvent être intégrées sans modification dans tout projet utilisant le framework .NET 4 ou supérieur, Silverlight, Windows 8 et supérieur, Windows Phone 8 et supérieur.

Les classes permettant la liaison avec l'utilisateur, en entrée ou en sortie, sont stockées dans un projet Windows de type **console** (à l'exception du chapitre Systèmes multi-agents présentant des applications graphiques WPF, *Windows Presentation Foundation*). Seul ce code doit être adapté en cas de changement de plateforme.

Les variables et méthodes sont nommées en **anglais**. Les paramètres des méthodes sont préfixés du symbole "_" et les constantes sont en majuscules.

La **visibilité** des classes, méthodes, propriétés et attributs a été limitée à ce qui était nécessaire, pour une meilleure sécurité de l'ensemble. Cependant, les règles de conception demandant un codage en couches ou selon le modèle MVVM (*Model View ViewModel*) n'ont volontairement pas été respectées. En effet, cela aurait rajouté des classes dans des projets qui restent de petite taille, et aurait diminué la lisibilité et la compréhension du code.

De même, les algorithmes présentés n'ont pas été optimisés lorsque ces optimisations allaient à l'encontre de la compréhension et de la lisibilité du code.

Bonne lecture à tous !

Introduction

1. Structure du chapitre

L'**intelligence artificielle** consiste à rendre intelligent un système artificiel, principalement informatique. Cela suppose qu'il existe une définition précise de l'**intelligence**. Or, ce n'est pas forcément le cas.

Cette introduction s'intéresse d'abord à l'intelligence chez les humains et à la façon de la définir. Ensuite est expliqué comment cette définition peut s'appliquer à d'autres formes de vie, que ce soient les animaux ou les végétaux, car si l'intelligence n'était liée qu'à l'humanité, il serait vain d'essayer de la recréer dans des systèmes artificiels.

Une fois posé le fait que l'intelligence peut se trouver en tout être vivant, nous verrons comment définir l'intelligence artificielle, ainsi que les grands courants de pensée que l'on y retrouve. Enfin, cette introduction se termine par un petit tour d'horizon des domaines d'application de celle-ci.

2. Définir l'intelligence

Il est important de comprendre tout d'abord ce qu'est l'**intelligence**. De nombreuses idées reçues circulent sur ce sujet, et peuvent gêner, voire rendre impossible, la compréhension du champ de l'intelligence artificielle.

Le terme d'intelligence vient du latin « intelligentia » qui signifie la faculté de comprendre et de mettre en relation des éléments entre eux.

L'intelligence est cependant multiple, et tous les auteurs actuels s'accordent sur le fait qu'il n'y a pas une mais des intelligences, et que chacun d'entre nous peut présenter des forces et/ou des faiblesses dans les différentes formes d'intelligence. La théorie des intelligences multiples, proposée initialement par Howard Gardner en 1983 (professeur à Harvard et travaillant sur les enfants en échec scolaire), liste sept formes d'intelligence, auxquelles deux nouvelles se sont ajoutées pour arriver à la liste actuelle des neuf formes d'intelligence :

- **L'intelligence logico-mathématique** : elle représente la capacité à travailler à l'aide de chiffres, à analyser des situations, à mettre au point des raisonnements. Elle est mise en avant chez les scientifiques, en particulier en physique et mathématiques.
- **L'intelligence visuo-spatiale** : elle indique la capacité à se représenter un objet ou un environnement en 3D, et est utilisée pour suivre une carte, se rappeler un chemin ou imaginer ce que donne une forme dans l'espace à partir de son plan. Elle est nécessaire par exemple aux artistes, aux architectes ou aux conducteurs de taxi.
- **L'intelligence verbo-linguistique** : il s'agit de la capacité à comprendre et à énoncer des idées par le langage. Elle requiert une bonne connaissance et maîtrise du vocabulaire, ainsi que de la syntaxe et des figures de style. Elle aide les avocats, les politiciens ou les auteurs.
- **L'intelligence intrapersonnelle** : elle est la capacité à avoir une image fidèle de soi, ce qui signifie pouvoir déterminer son état émotionnel, ses envies, ses forces et ses faiblesses.
- **L'intelligence interpersonnelle** : elle est la capacité à comprendre les autres et à réagir de la façon adéquate. Elle est donc liée à la notion d'empathie, à la tolérance, à la sociabilité. Mais elle peut aussi permettre de manipuler et est ainsi très utilisée par les leaders de secte. Elle a aussi inspiré des techniques commerciales et de négociation.
- **L'intelligence corporelle/kinesthésique** : elle est la capacité à avoir une représentation mentale de son corps dans l'espace et à pouvoir mener un mouvement particulier. Très utilisée chez les athlètes, c'est elle qui permet d'avoir le bon geste au bon moment. Elle est utilisée dans les travaux manuels et de précision (par exemple pour un chirurgien), mais permet aussi l'expression corporelle des émotions et est à ce titre nécessaire aux danseurs ou aux acteurs.

- **L'intelligence naturaliste** : c'est la capacité à trier, organiser et hiérarchiser les objets qui nous entourent. Elle permet ainsi de définir des espèces, des sous-espèces, ou de construire des classifications. Elle est par exemple très utilisée par les botanistes, les paléontologues ou les biologistes.
- **L'intelligence musicale** : elle est la capacité à reconnaître les mélodies, les notes et les harmonies, ou à les créer. Elle est ainsi nécessaire aux compositeurs et aux chanteurs et s'exprime chez tous les mélomanes.
- **L'intelligence existentielle ou spirituelle** : elle est la capacité à se poser des questions sur le sens de la vie, sur notre but. Elle se rapproche de notre notion de moralité. Elle n'est pas forcément liée à la notion de religion mais plus à notre positionnement par rapport au reste de l'univers.

De nombreux tests ont essayé de mesurer l'intelligence, le plus connu étant le **test de Q.I.** (quotient intellectuel). Ils sont cependant très critiqués. En effet, ils ne peuvent pas mesurer toute l'amplitude des formes d'intelligence et se concentrent principalement sur les intelligences logico-mathématique et visuo-spatiale (même si l'intelligence verbo-linguistique est en partie testée). Toutes les autres formes d'intelligence sont ignorées.

De plus, les principaux tests de Q.I. que l'on trouve sont biaisés par l'expérience : il suffit de faire et refaire plusieurs fois des entraînements sur ces tests pour obtenir des résultats significativement améliorés. Pour autant, est-on devenu plus intelligent ? La répétition et le bachotage ne donnent que des habitudes, des réflexes et des bonnes pratiques, pour ces types de problèmes précisément, mais cet apprentissage n'est pas valorisable.

Le système scolaire lui-même met principalement en avant ces trois formes d'intelligence (logico-mathématique, visuo-spatiale et verbo-linguisitique). Les autres formes d'intelligence sont délaissées, et étudiées dans les matières dites "annexes" (sport, musique, technologie...) et certaines ne sont pas du tout abordées (intelligences intra, interpersonnelle et existentielle).

Il faut donc accepter que l'intelligence n'est pas facilement mesurable, ni facilement définissable, car elle couvre de trop nombreux domaines. De plus, certains types sont souvent sous-estimés voire ignorés.

La meilleure définition est donc aussi la plus vaste : **l'intelligence est la capacité à s'adapter**. Elle permet ainsi de résoudre les problèmes auxquels on est confronté.

Cette définition a, par exemple, été donnée en 1963 par Piaget (biologiste de formation et psychologue).

3. L'intelligence du vivant

L'intelligence est trop souvent liée de près à celle de l'humain. En effet, les Hommes ont cherché à se montrer supérieurs aux animaux, et tout ce qui pouvait les différencier était bon à prendre pour se distinguer des « bêtes ». Ce terme est d'ailleurs très significatif : il désigne à la fois les animaux et les personnes étant considérées comme ne possédant que peu d'intelligence.

Pourtant, la définition de l'intelligence comme capacité à s'adapter permet de prendre en compte de nombreux comportements que l'on trouve chez les animaux, et même plus globalement chez les êtres vivants.

Quand on parle **« intelligence du vivant »**, on pense souvent aux grands singes (capables d'apprendre le langage des signes et de communiquer grâce à lui), aux chiens ou aux dauphins. On peut aussi citer le cas de Hans le malin, un cheval qui « savait » compter et répondait par des coups de sabot sur le sol (par exemple à la question « Combien font 3 plus 4 ? », il tapait 7 fois du pied). En réalité, il arrivait à détecter les micromouvements sur les visages du public pour savoir quand il devait s'arrêter : il avait ainsi adapté son comportement à son environnement pour obtenir des friandises et des caresses.

On peut aussi parler des animaux montrant une **intelligence collective** remarquable. Il y a par exemple les termites qui sont capables de construire des nids immenses et climatisés, faits de multitudes de couloirs et de chambres. Les fourmis sont un autre très bon exemple avec la présence de rôles : reine, ouvrières, nourrices, gardiennes, combattantes et même éleveuses, étant donné que certaines espèces « élèvent » des pucerons et les protègent des coccinelles pour ensuite les « traire » et manger le miellat produit.

Chez les abeilles, l'**intelligence linguistique** est très forte. En effet, lorsqu'une abeille rentre à la ruche après avoir trouvé une source de pollen, elle va l'indiquer à ses pairs via une danse. En fonction de la forme et de la vitesse de celle-ci, les autres abeilles en déduisent la distance. En regardant l'angle fait avec le soleil, elles ont alors l'indication de direction. La danse en elle-même peut les renseigner sur la source de nourriture (type de fleur par exemple).

Mais l'intelligence est en réalité présente pour toutes les formes vivantes. De nombreuses espèces végétales se sont adaptées pour attirer certaines proies (comme les plantes carnivores) ou les insectes qui vont disséminer leur pollen. Au contraire, certaines se protègent via des sucs toxiques ou des épines.

D'autres enfin attirent les prédateurs naturels de leurs propres prédateurs. Par exemple, certaines plantes de la famille des Acacias attirent les fourmis pour se protéger des herbivores. Elles appliquent ainsi le fameux dicton « les ennemis de mes ennemis sont mes amis ».

On peut aller encore plus loin. Certains champignons ont développé des stratégies très complexes pour survivre et se reproduire, et certains présents dans les forêts amazoniennes du Brésil sont ainsi capables d'utiliser des fourmis comme hôte (en étant ingérés par exemple sous forme de spores), d'en prendre le contrôle temporaire (via des sécrétions attaquant les fonctions cérébrales, ce qui conduit la fourmi à s'éloigner du nid, à grimper le plus haut possible et à finalement s'attacher à une feuille), et de s'en servir comme source nutritive puis comme point de départ d'une nouvelle envolée de spores.

Certes, tous les individus d'une même espèce n'ont pas le même niveau d'intelligence, mais il est impossible de nier que l'on peut trouver des formes d'intelligence dans les espèces vivantes.

4. L'intelligence artificielle

La nature présente de nombreux cas d'intelligence : elle n'est pas spécifique à l'homme. En fait, elle n'est même pas spécifique au vivant : tout système qui pourrait s'adapter pour donner une réponse adéquate à son environnement pourrait être considéré comme intelligent. On parle alors d'**intelligence artificielle** (I.A.). Le terme en lui-même a été créé par John McCarthy en 1956 (l'I.A. a une histoire riche et longue).

Le domaine de l'intelligence artificielle est très vaste et peut couvrir de nombreuses techniques différentes. Les capacités de calcul toujours plus importantes des ordinateurs, une meilleure compréhension de certains processus naturels liés à l'intelligence et les progrès des chercheurs dans les sciences fondamentales ont permis de grandes avancées.

Pour autant, toutes les facultés que l'on peut donner à un ordinateur ne sont pas considérées comme faisant partie de l'intelligence artificielle. Ainsi, un ordinateur qui peut résoudre des équations complexes dans un temps très court (beaucoup plus que ce que pourrait un humain) n'est cependant pas considéré comme intelligent.

Comme pour les humains (ou les animaux), il existe des tests pour déterminer si on peut considérer que le programme est, ou non, intelligent. Le plus connu est le **test de Turing** (décrit en 1950 par Alan Turing), qui consiste à faire communiquer un testeur humain avec deux écrans. Derrière l'un de ces écrans, c'est un autre humain qui écrit. Derrière le deuxième, le programme testé s'exécute. On demande, après une phase où le testeur discute avec les deux systèmes, de déterminer quel était l'humain. S'il n'arrive pas à différencier la machine de l'humain, alors le test est réussi.

Ce test subit, comme pour les tests de Q.I., de nombreuses critiques. Tout d'abord, il ne s'applique pas à toutes les formes d'intelligence et ne peut pas tester tous les programmes (uniquement ceux destinés à la communication). De plus, un programme « non intelligent » qui ne ferait que reprendre en partie les phrases dites sans les comprendre peut réussir ce test, comme c'est le cas du programme ELIZA créé en 1966.

Celui-ci reconnaît des structures de phrases pour en extraire les mots importants et les réutilise dans les questions suivantes. Par exemple, à la phrase « J'aime le chocolat », le programme répondra « Pourquoi dites-vous que vous aimez le chocolat ? ». On retrouve ce logiciel comme psychologue dans l'éditeur de texte Emacs.

Enfin, un programme trop intelligent qui pourrait répondre à toutes les questions de manière correcte ou qui ne ferait aucune faute d'orthographe, de grammaire ou simplement de frappe serait vite reconnu et échouerait le test.

Il est donc difficile de définir exactement l'intelligence artificielle : il faut surtout que **la machine donne l'impression d'être intelligente** lorsqu'elle résout un problème, en mimant par exemple le comportement humain ou en mettant en place des stratégies plus souples que celles permises par de la programmation classique. Là encore, on retrouve une notion d'**adaptabilité**.

Il est par contre important de noter qu'il n'y a aucune notion de technologie, de langage ou de domaine d'application dans cette définition. Il s'agit d'un champ très vaste, que l'on peut néanmoins séparer en deux grands courants :

– **L'approche symbolique** : l'environnement est décrit le plus précisément possible, ainsi que les lois qui s'y appliquent, et c'est au programme de choisir la meilleure option. C'est l'approche utilisée dans les systèmes experts ou la logique floue par exemple.
– **L'approche connexionniste** : on donne au logiciel un moyen d'évaluer si ce qu'il fait est bien ou non, et on le laisse trouver des solutions seul, par émergence. Cette approche est celle des réseaux de neurones ou des systèmes multi-agents.

Ces deux approches ne sont cependant pas totalement contradictoires, et peuvent donc être complémentaires pour résoudre certains problèmes : on peut par exemple partir d'une base symbolique qui pourra être complétée par une approche connexionniste.

5. Domaines d'application

L'intelligence artificielle est souvent associée à la **science-fiction**. On la retrouve ainsi dans de nombreux films ou livres, comme l'ordinateur HAL 9000 de l'Odyssée de l'espace de Stanley Kubrick (1968). Malheureusement (pour nous, humains), ces I.A. ont la fâcheuse tendance à se rebeller et/ou à vouloir soumettre les hommes, parfois même « pour leur bien » comme dans le film I. Robot d'Alex Proyas (2004).

Actuellement, l'intelligence artificielle est effectivement utilisée dans le monde de la **robotique**, pour permettre aux robots d'interagir de manière plus souple avec les humains qu'ils doivent aider. Les tâches à faire sont parfois très simples, comme le lavage du sol, ou beaucoup plus complexes pour les « robots de compagnie » qui doivent aider dans la vie de tous les jours des personnes qui n'ont plus toutes leurs capacités (par exemple les personnes âgées ou en situation de handicap). De nombreux travaux sont en cours dans ce domaine, et les possibilités sont quasi infinies.

Les **militaires** l'ont d'ailleurs bien compris : de nombreux robots sont commandés ou subventionnés sur leurs fonds de recherche. On parle de drones intelligents qui pourraient chercher des ennemis sur des zones de combat, de soldats mécaniques, mais aussi de robots qui permettraient de retrouver et de sauver les victimes de catastrophes naturelles.

Un autre grand domaine de l'intelligence artificielle est le **jeu vidéo**. En effet, pour avoir un jeu réaliste, il est nécessaire que les personnages (ennemis ou alliés) aient un comportement qui paraisse le plus cohérent possible aux yeux des joueurs. Dans un jeu type Metal Gear, un ennemi qui foncerait sur vous avec un petit couteau dans une zone dégagée n'est ainsi pas réaliste, alors que si celui-ci se faufile dans les recoins et vous attaque par derrière, il parait « vivant ». De plus, si les monstres de Super Mario ont des chemins prédéfinis, ce genre de stratégie ne peut s'appliquer dans des jeux où l'immersion est importante.

Même dans le titre Pac-Man, les différents fantômes sont chacun dotés d'une intelligence artificielle pour contrôler leurs mouvements : Blinky (le fantôme rouge) essaie d'atteindre la case dans laquelle se trouve actuellement le joueur ; Pinky (rose) essaie de se placer quatre cases devant lui (pour le piéger) alors que Clyde (orange) alterne entre essayer d'attraper le personnage ou s'en éloigner (ce qui ne le rend pas très dangereux). Enfin, Inky (bleu) essaie de bloquer le joueur à la manière de Pinky mais en utilisant en plus la position de Blinky, pour le « prendre en sandwich ».

Si la robotique et les jeux vidéo sont des domaines évidents d'application de l'intelligence artificielle, ils ne sont cependant que la partie immergée de l'iceberg. De nombreux autres domaines utilisent l'I.A., du milieu **bancaire** à la **médecine** en passant par l'informatique industrielle.

En effet, les systèmes experts qui permettent de prendre une décision grâce à des règles plus ou moins évoluées sont utilisés pour détecter des fraudes (par exemple fraude à la carte bancaire) ou pour détecter des modifications de comportement (c'est ainsi que l'on peut vous proposer des contrats téléphoniques ou d'énergie plus adaptés quand vous changez de mode de vie). Ils sont aussi très utilisés en médecine pour aider au diagnostic, en fonction des symptômes du patient, et ce de manière plus rapide et plus complète qu'un médecin (même si ce dernier reste le seul à prendre des décisions).

L'I.A. peut se retrouver dans des domaines de la vie courante comme l'**informatique personnelle** : Clippy, le « trombone » de la suite Microsoft Office, en est un bon exemple. Le courrier postal passe lui aussi par des machines dotées d'intelligence artificielle. En effet, les adresses manuscrites sont lues et reconnues, puis traduites et marquées sur les enveloppes sous la forme d'un code-barre à l'encre orange phosphorescente. Les vitesses de lecture sont impressionnantes avec la possibilité d'encoder jusqu'à 50 000 lettres par heure, ce qui fait près de 14 lettres à la seconde !

L'**informatique industrielle** utilise aussi fortement l'I.A. par exemple en logistique, pour optimiser les trajets des camions de livraison ou le remplissage de ceux-ci. Le rangement dans les entrepôts peut lui aussi être amélioré grâce à des algorithmes d'intelligence artificielle. Les pièces peuvent être rendues plus efficaces en changeant leur forme, et les circuits imprimés sont optimisés pour limiter la quantité de « ponts » ou de matière conductrice.

6. Synthèse

L'intelligence est un concept difficile à définir précisément, car celle-ci peut prendre de nombreuses formes. Il est tout aussi difficile de la mesurer et les tests de Q.I. sont biaisés. Elle peut se résumer comme la capacité d'adaptation à son environnement pour y résoudre les problèmes qui se présentent.

Le règne animal est donc lui aussi doté d'intelligence, certes différente dans ses exemples, mais bien présente. Plus généralement, tous les êtres vivants, par leur adaptation à leur environnement et la création de stratégies de survie complexes font preuve d'intelligence.

Celle-ci peut être « implantée » dans des machines. L'intelligence artificielle revient donc à doter un système d'un mécanisme lui permettant de simuler le comportement d'un être vivant, de mieux le comprendre ou encore d'adapter sa stratégie aux modifications de l'environnement. Là encore, il n'est pas possible réellement de déterminer si un logiciel présente une forme d'intelligence, les tests type « tests de Turing » possédant, tout comme les tests de Q.I., des limites.

Les technologies, les langages et les algorithmes sont aussi nombreux que les domaines d'application, et l'I.A. n'est pas réservée à la robotique ou aux jeux vidéo. En effet, on peut la retrouver dans quasiment tous les domaines informatisés. Elle nous entoure, sans même que l'on s'en rende compte, et peut améliorer quasiment tout type de logiciel.

Il s'agit d'un domaine en plein essor et les capacités grandissantes des ordinateurs ont permis de mettre au point des algorithmes jusqu'alors impossibles. Sans aucun doute, l'I.A. va faire partie de notre futur et il est donc important que tout développeur ou plus généralement informaticien comprenne les mécanismes sous-jacents pour pouvoir les appliquer.

Chapitre 1
Systèmes experts

1. Présentation du chapitre

Bien souvent, on aimerait qu'un ordinateur soit capable de nous donner une information que l'on ne connaît pas à partir de faits connus.

Les êtres humains eux-mêmes ne savent pas toujours déduire des faits d'autres faits qui leur sont connus et nécessitent l'aide d'un **expert**. Par exemple dans le cas d'une panne automobile, la majorité des personnes ne peut pas déterminer l'origine de celle-ci et se tourne alors vers le garagiste (l'expert). Celui-ci, grâce à ses connaissances, va pouvoir trouver la panne (et la réparer, normalement).

Beaucoup d'emplois sont constitués de ces experts. Les médecins, les assureurs ou les agents immobiliers n'en sont que quelques exemples.

L'intelligence artificielle peut nous aider, en créant un programme informatique appelé **système expert** qui jouera le rôle de ce professionnel. Dans certains cas comme la médecine, cet outil pourra devenir une aide au spécialiste lui-même car le domaine étudié est très vaste. Il est en effet rare que l'expert humain puisse être complètement remplacé, et il sera souvent là en appui pour confirmer la conclusion du système, en lui faisant tout de même gagner un temps précieux. Dans d'autres cas, le système donnera un premier diagnostic, qui ne sera complété par une personne physique que dans le cas où la panne n'est pas connue de la base de données utilisée (comme pour les pannes très rares par exemple). Dans la majorité des cas, le système expert sera suffisant.

Ce chapitre présente donc les systèmes experts en commençant par leurs concepts. Un exemple sert de fil rouge, pour bien comprendre comment toutes les parties d'un système expert communiquent.

Ensuite les grands domaines d'application sont présentés. L'implémentation de systèmes plus ou moins complexes, en C# et en Prolog, suit cette présentation. Enfin est abordé le cas dans lequel les connaissances du domaine présentent un degré d'incertitude, et les modifications nécessaires pour gérer ceux-ci sont alors expliquées.

Ce chapitre se termine par une petite synthèse.

2. Exemple : un système expert en polygones

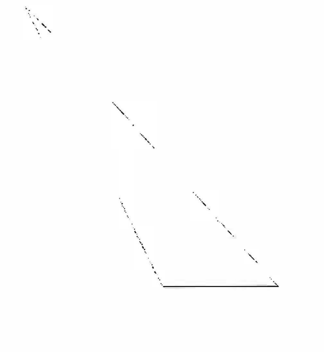
Cette section permet de voir le fonctionnement détaillé d'un système expert dont le but est de déterminer le nom d'un polygone (par exemple un rectangle) en fonction de caractéristiques sur la forme (le nombre de côtés, les angles droits...). Un petit rappel de géométrie commence donc ce chapitre.

Un polygone est défini comme une forme géométrique possédant au moins trois côtés. L'**ordre** d'un polygone correspond au nombre de ses côtés.

2.1 Triangles

Si l'ordre d'un polygone vaut 3, il possède donc trois côtés et il s'agit d'un triangle. Celui-ci peut être quelconque, rectangle, isocèle, rectangle isocèle ou équilatéral.

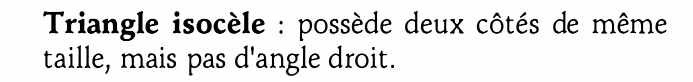
Les figures suivantes rappellent les particularités de chacun.



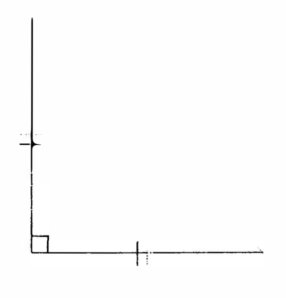
Triangle quelconque : possède trois côtés de tailles différentes et aucun angle droit.



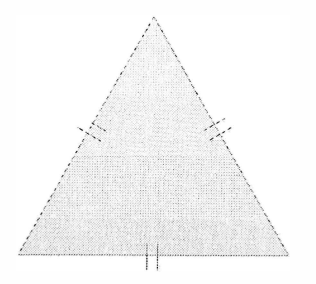
Triangle rectangle : possède trois côtés de tailles différentes et un angle droit.



Triangle isocèle : possède deux côtés de même taille, mais pas d'angle droit.



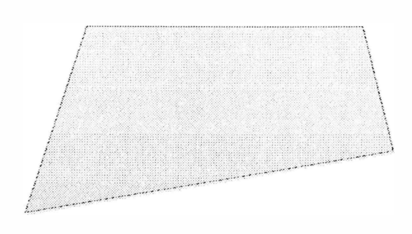
Triangle rectangle isocèle : cumule les deux côtés égaux du triangle isocèle et l'angle droit du triangle rectangle.



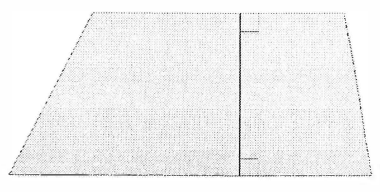
Triangle équilatéral : possède trois côtés de même taille (et ne peut pas posséder d'angle droit).

2.2 Quadrilatères

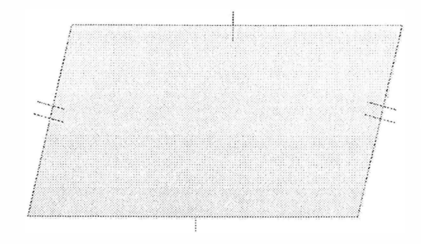
Lorsque l'ordre d'un polygone vaut 4, on parle de quadrilatère. Celui-ci peut être quelconque, ou il peut s'agir d'un trapèze, d'un parallélogramme, d'un losange, d'un rectangle ou d'un carré. Les figures suivantes présentent ces différents quadrilatères et leurs propriétés.



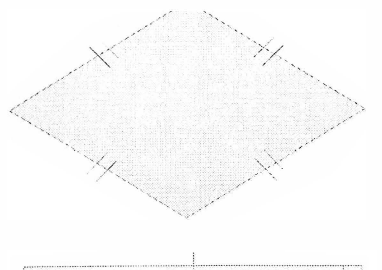
Quadrilatère quelconque : possède quatre côtés non parallèles.



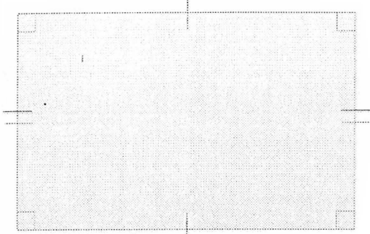
Trapèze : possède deux côtés (et uniquement deux) parallèles.



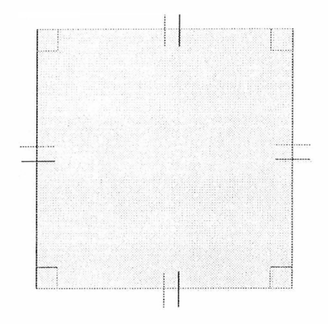
Parallélogramme : possède quatre côtés parallèles deux à deux. Possède aussi des côtés opposés de même taille.



Losange : parallélogramme dont les côtés sont tous de la même taille.



Rectangle : parallélogramme possédant des angles droits.



Carré : parallélogramme cumulant les côtés de même taille du losange et les angles droits du rectangle.

2.3 Autres polygones

Lorsque l'ordre est supérieur à 4, le nom du polygone est donné dans le tableau suivant pour les cas les plus courants. On parle de polygone régulier lorsque tous les côtés sont de la même taille.

Ordre	Nom du polygone
5	Pentagone
6	Hexagone

Ordre	Nom du polygone
8	Octogone
10	Décagone
12	Dodécagone
20	Icosagone

Remarque

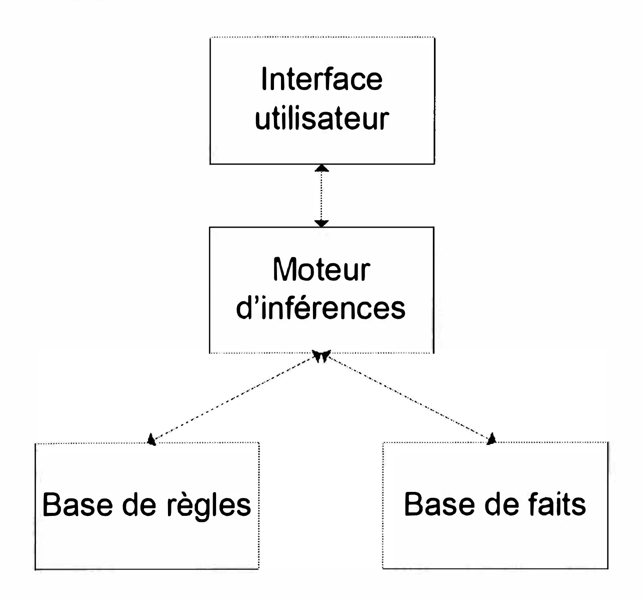
Tous les polygones, quel que soit leur ordre, possèdent un nom particulier. Ainsi un polygone d'ordre 26 s'appelle un « hexaicosagone ». Cependant, seuls les noms les plus courants sont rappelés dans le tableau précédent.

3. Contenu d'un système expert

Un système expert est constitué de différentes parties liées entre elles :

- Une **base de règles** qui représente les connaissances de l'expert.
- Une **base de faits** qui représente les connaissances actuelles du système sur un cas précis.
- Un **moteur d'inférences** pour appliquer les règles.
- Une **interface** avec l'utilisateur.

Le schéma suivant indique les liens entre ces différentes parties, qui seront détaillées ensuite.



3.1 Base de règles

Un système expert contient un ensemble de règles nommé **base de règles**. Celles-ci représentent les connaissances de l'expert sur le domaine.

Ces règles sont toujours de la forme :

```
SI (ensemble de conditions) ALORS nouvelle connaissance
```

Les conditions d'application d'une règle sont appelées les **prémisses**. Il peut y avoir plusieurs prémisses, elles sont alors reliées par une coordination ET, signifiant qu'elles doivent toutes être vraies pour que la règle s'applique.

Les nouvelles connaissances sont appelées **conclusions**.

Pour notre système expert sur les formes géométriques, voici les règles concernant les triangles :

```
SI (ordre vaut 3) ALORS c'est un triangle
SI (triangle ET 1 angle droit) ALORS c'est un triangle rectangle
```

```
SI (triangle ET 2 côtés de même taille) ALORS c'est un triangle isocèle
SI (triangle rectangle ET triangle isocèle) ALORS c'est un triangle
rectangle isocèle
SI (triangle ET côtés tous égaux) ALORS c'est un triangle équilatéral
```

Il existerait d'autres règles pour les quadrilatères et les polygones d'ordre supérieur. On voit que très rapidement le nombre de règles peut être important.

De plus, selon le système utilisé, chaque règle doit suivre une syntaxe précise et imposée.

En particulier, les prémisses et conclusions peuvent être demandées sous la forme *attribut(valeur)*, par exemple ordre(3) ou angleDroit(1). La règle du triangle rectangle dans une telle représentation pourrait être :

```
SI (ordre(3) ET angleDroit(1)) ALORS polygone(TriangleRectangle)
```

Remarque

Lorsque le système expert est construit de toutes pièces, il est possible de choisir le format des règles de manière à ce qu'il se rapproche le plus possible du domaine étudié. Cela facilitera les étapes de mise en place et la création du moteur.

3.2 Base de faits

Les prémisses d'une règle peuvent être de deux types :

– Des connaissances sur le problème fournies par l'utilisateur du système : ce sont les **entrées**.

– Des connaissances issues de l'application de règles : ce sont les **faits inférés**.

Ces deux types de connaissances doivent être enregistrés dans une **base de faits** qui contient donc toutes les informations sur le problème, quelle que soit leur origine. Lorsqu'on lance un système expert, la base ne contient initialement que les connaissances de l'utilisateur (les entrées) et se remplit petit à petit des faits inférés.

Supposons que nous ayons affaire à une forme d'ordre 4, possédant 4 côtés parallèles, de même taille et 4 angles droits. Ce sont nos connaissances initiales.

À ces faits en entrée va se rajouter le fait que la figure est un quadrilatère (elle est d'ordre 4) et un parallélogramme (un quadrilatère avec 4 côtés parallèles), qu'il s'agit plus précisément d'un losange (un parallélogramme dont les 4 côtés sont de même taille) et qu'il s'agit aussi d'un rectangle (un parallélogramme avec des angles droits). Enfin, on ajoute le fait qu'il s'agit d'un carré (car il s'agit d'un losange et d'un rectangle).

Généralement, c'est le dernier fait ajouté qui intéresse vraiment l'utilisateur : il s'agit du **but du programme**.

Cependant, on peut aussi concevoir un système expert qui permet de savoir si un fait est vrai ou non. Dans ce cas, on regarde si le fait donné est dans la base de faits après application des règles.

Dans l'exemple de notre quadrilatère précédent, au lieu de demander quel est le nom de ce polygone (il s'agit d'un carré), on pourrait demander s'il s'agit d'un losange ou s'il s'agit d'un triangle rectangle. On obtiendrait une réponse affirmative dans le premier cas (un carré est un losange particulier), et négative dans le deuxième (un carré n'est pas un triangle).

3.3 Moteur d'inférences

Le **moteur d'inférences** (ou **système d'inférences**) est le cœur du système expert.

Le moteur va permettre de sélectionner et d'appliquer les règles. Cette tâche n'est pas forcément aisée car il peut exister des milliers de règles. De plus, il ne sert à rien d'appliquer une règle déjà utilisée précédemment ou qui ne correspond pas du tout au problème à résoudre.

Par exemple, si on créé un système expert permettant de reconnaître la faune et la flore d'une forêt et que l'on cherche à savoir de quelle famille est un insecte trouvé, il ne sert à rien d'essayer d'appliquer les règles concernant les arbres et les buissons.

C'est aussi le moteur d'inférences qui va ajouter les nouveaux faits à la base de faits, ou y accéder pour vérifier qu'un fait est déjà connu. L'ajout des faits que l'on sait faux est tout aussi intéressant que celui des faits que l'on sait justes. En effet, savoir qu'une forme n'est pas un quadrilatère permet d'éliminer plusieurs règles d'un coup. Dans un système plus complexe, savoir que l'insecte recherché ne possède pas d'ailes est aussi très instructif et peut limiter les tentatives pour retrouver l'espèce cherchée.

■Remarque

Une analogie simple est le jeu du « Qui est-ce ? ». Dans ce jeu, il faut retrouver un personnage parmi plusieurs en ne posant que des questions dont les réponses sont oui ou non (par exemple « le personnage porte-t-il un chapeau ? »). Que la réponse soit positive ou non, cela apporte de la connaissance. En effet, savoir qu'il ne possède pas de chapeau permet d'éliminer les personnages en possédant, tout comme le fait de savoir qu'il porte des lunettes permet de ne garder que ceux qui en portent.

Enfin, le moteur doit pouvoir prendre une décision importante : celle de s'arrêter pour présenter à l'utilisateur la réponse à sa question. Il doit donc savoir quand un but est atteint, ou quand il ne le sera jamais.

L'avantage des systèmes experts sur de nombreuses autres techniques d'intelligence artificielle est leur capacité à expliquer le raisonnement suivi : les moteurs implémentent donc souvent un mécanisme permettant de retrouver toutes les règles utilisées pour arriver à un fait donné. L'utilisateur obtient le raisonnement en plus du résultat, ce qui peut être très important dans certains domaines.

De nombreux moteurs d'inférences sont disponibles ou peuvent être codés de toutes pièces, et ce dans n'importe quel langage de programmation. Certains langages ont cependant été créés dans le but d'implémenter des moteurs d'inférences. Ils correspondent à la famille des langages de **programmation logique**, dont fait partie Prolog. Cependant la création d'un moteur d'inférences dans un langage objet comme le C# est aussi possible. Enfin, il existe des ponts entre ces langages, permettant d'utiliser un moteur d'inférences en Prolog dans un code C# par exemple.

3.4 Interface utilisateur

Le dernier élément d'un système expert est l'**interface utilisateur**. En effet, il faut que l'utilisateur puisse simplement entrer les données qu'il possède, soit en une seule fois avant de lancer le processus, soit au fur et à mesure des besoins du moteur d'inférences.

Comme pour tout logiciel, si l'interface n'est pas agréable à utiliser ou si elle est trop complexe, voire contre-intuitive, le système sera peu ou pas utilisé.

De plus, dans un système expert, il est primordial que les choix dont dispose l'utilisateur soient clairs, pour qu'il puisse répondre correctement aux questions posées, sans quoi le résultat retourné serait faux.

Dans le cas du système expert sur les polygones, il y a peu de chances d'erreurs, car les règles sont basées sur le nombre de côtés, les angles droits, les côtés parallèles et les tailles des segments. Pourtant, une interface demandant l'ordre du polygone sans préciser qu'il s'agit du nombre de côtés ne serait pas adéquate.

Dans le cas de la reconnaissance d'insectes, si le logiciel demande si celui-ci possède des cerques ou des cornicules, il y a peu de chances qu'un utilisateur non entomologiste puisse répondre correctement. Par contre, si on lui demande si le corps se finit par deux longues pointes (cerques, que l'on retrouve par exemple chez les perce-oreilles) ou deux très courtes pointes (cornicules, présentes chez le puceron), il y aura moins d'erreurs ou de confusions. Une photo ou un dessin serait encore plus parlant pour l'utilisateur.

Il est donc important de travailler sur l'ergonomie du système expert avec de futurs utilisateurs ou des représentants des utilisateurs, de manière à savoir quels termes seraient les plus adaptés, ainsi que la disposition des écrans, pour limiter les erreurs.

4. Types d'inférences

Les moteurs d'inférences peuvent enchaîner les règles de différentes façons : c'est ce que l'on appelle le **chaînage**. Les deux principaux chaînages sont le chaînage avant et le chaînage arrière, mais il existe des moteurs possédant un chaînage mixte.

4.1 Chaînage avant

4.1.1 Principe

Un moteur à **chaînage avant** est aussi appelé un moteur à inférences **dirigé par les données**.

Dans ce mode de chaînage, on part des données disponibles en base de faits, et on teste pour chaque règle si elle peut s'appliquer ou non. Si oui, on l'applique et on rajoute la conclusion à la base de faits.

Le moteur explore donc toutes les possibilités, jusqu'à trouver le fait recherché ou jusqu'à ne plus pouvoir appliquer de nouvelles règles.

Ce mode de chaînage est celui proposé par défaut dans des langages de type CLIPS (*C Language Integrated Production System*), spécialisés dans la construction de systèmes experts.

4.1.2 Application à un exemple

Dans le cas de notre système expert sur les polygones, supposons que nous partions des faits suivants :

– L'ordre vaut 3.

– Il y a un angle droit.

– Deux côtés sont de même taille.

On commence par appliquer la règle suivante, qui ajoute dans la base de faits que la forme est un triangle :

```
SI (ordre vaut 3) ALORS c'est un triangle
```

On peut ensuite en déduire que c'est un triangle rectangle grâce à la règle suivante :

```
SI (triangle ET 1 angle droit) ALORS c'est un triangle rectangle
```

De même, on sait que c'est un triangle isocèle :

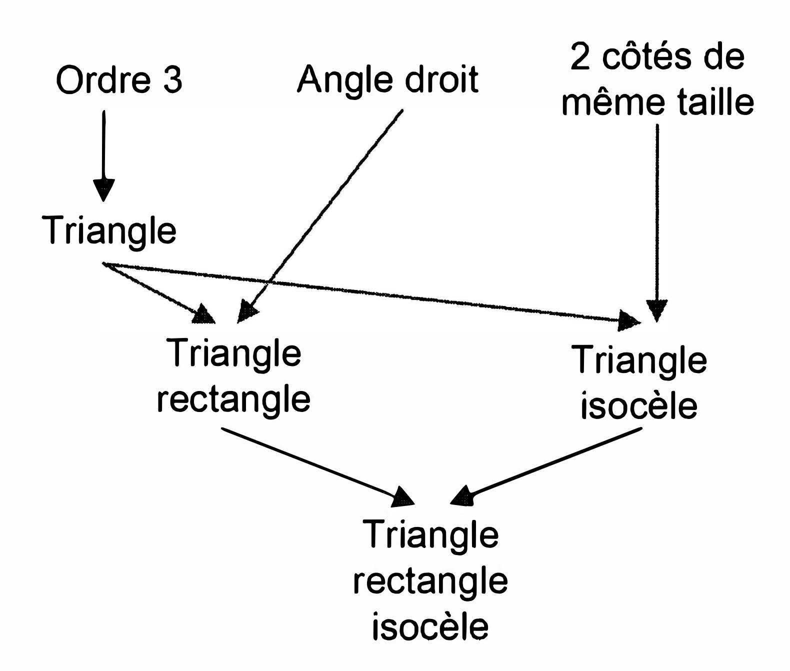
```
SI (triangle ET 2 côtés de même taille) ALORS c'est un triangle isocèle
```

Enfin, en sachant qu'il s'agit d'un triangle rectangle et d'un triangle isocèle, on peut appliquer :

```
SI (triangle rectangle ET triangle isocèle) ALORS c'est un triangle
rectangle isocèle
```

On rajoute donc enfin le fait qu'il s'agit d'un triangle rectangle isocèle. Comme il n'y a plus de règles applicables, le moteur d'inférences s'arrête. L'utilisateur est informé que son polygone est un triangle rectangle isocèle.

On peut résumer la logique du moteur par le schéma suivant :



4.2 Chaînage arrière

4.2.1 Principe

Les moteurs d'inférences à **chaînage arrière** sont aussi dits **dirigés par le but**.

Ce coup-ci, on part des faits que l'on souhaiterait obtenir et on cherche une règle qui pourrait permettre d'obtenir ce fait. On rajoute alors toutes les prémisses de cette règle dans les nouveaux buts à atteindre.

On réitère, jusqu'à ce que les nouveaux buts à atteindre soient présents dans la base de faits. Si un fait est absent de la base de faits ou prouvé comme faux, alors on sait que la règle ne peut pas s'appliquer. Ces moteurs ont donc un mécanisme (le **backtracking**) leur permettant de passer à une nouvelle règle, qui serait un nouveau moyen de prouver le fait.

Si plus aucune règle ne peut mener au but recherché, alors celui-ci est considéré comme faux.

Ce mode de chaînage est celui présent par défaut dans le langage Prolog, dédié aux systèmes experts.

4.2.2 Application à un exemple

On reprend l'exemple précédent, à savoir un polygone pour lequel :

- L'ordre vaut 3.
- Il y a un angle droit.
- Deux côtés sont de même taille.

On demande au logiciel si le triangle est rectangle isocèle. C'est notre premier but. Le moteur recherche une règle permettant d'obtenir ce fait, il n'y en a qu'une :

```
SI (triangle rectangle ET triangle isocèle) ALORS c'est un triangle
rectangle isocèle
```

Le moteur ajoute donc les buts « triangle rectangle » et « triangle isocèle » à sa liste de buts. Il commence par chercher une règle permettant de prouver qu'il s'agit d'un triangle rectangle. Là encore, nous n'avons qu'une règle :

```
SI (triangle ET un angle droit) ALORS c'est un triangle rectangle
```

Il obtient ainsi deux nouveaux buts : s'agit-il d'un triangle et a-t-il un angle droit ? La présence d'un angle droit est déjà en base de faits, ce but est donc atteint. Pour le triangle, il a besoin de la règle suivante :

```
SI (ordre vaut 3) ALORS c'est un triangle
```

La base de faits précise que l'ordre a une valeur de 3, la règle est donc prouvée. Le fait triangle peut ainsi être ajouté à la base de faits et enlevé des buts à atteindre, tout comme le fait « triangle rectangle ». Il ne lui reste alors plus que « triangle isocèle » à prouver.

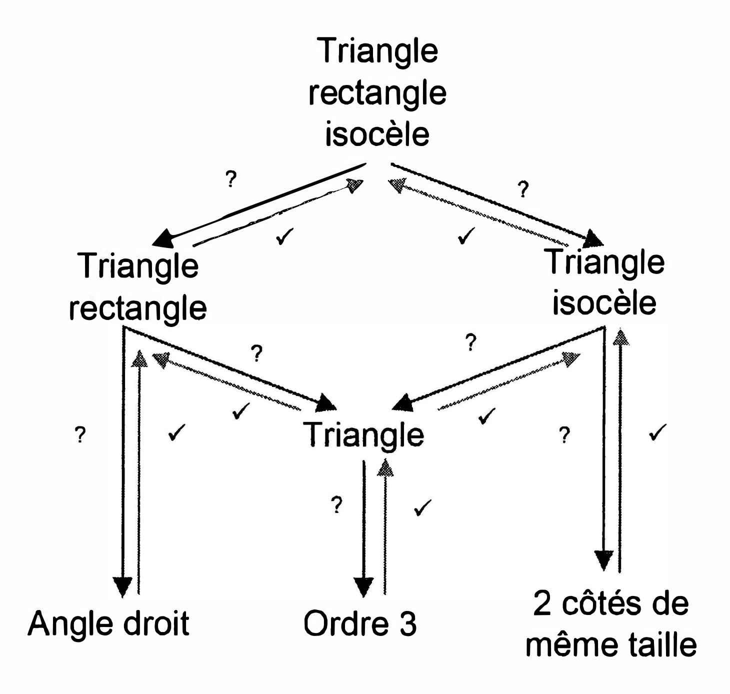
De la même façon, il cherche une règle possédant ce but :

```
SI (triangle ET 2 côtés de même taille) ALORS c'est un triangle isocèle
```

Le fait que la forme soit un triangle est déjà en base de faits (on l'a obtenu juste avant) et la présence de deux côtés de même taille aussi. On rajoute que c'est un triangle isocèle.

Le programme finit par retourner à son but initial, à savoir s'il s'agissait d'un triangle rectangle isocèle. Comme les faits « triangle rectangle » et « triangle isocèle » sont maintenant prouvés, il peut en conclure que oui, la forme est un triangle rectangle isocèle, et l'indiquer à l'utilisateur.

La logique est donc ce coup-ci la suivante : on part du but à atteindre et on essaie de prouver que les prémisses sont vraies.



4.3 Chaînage mixte

Chaque mode de chaînage a ses avantages et ses inconvénients :

– Le chaînage avant permet de découvrir en permanence de nouveaux faits, mais il risque d'appliquer et de tester de nombreuses règles qui ne concernent pas l'information recherchée par l'utilisateur. Il est donc plus adapté à l'exploration.

– Le chaînage arrière permet de se concentrer sur un but précis (ou plusieurs buts), mais il va tester de nombreuses possibilités qui seront finalement démontrées comme fausses. Ainsi, il va essayer de prouver des règles qui ne pourront pas l'être. Sa gestion est aussi plus complexe (car il doit gérer la liste des buts et permettre le backtracking).

Un mélange des deux chaînages a alors été proposé : le **chaînage mixte**. Dans ce nouveau chaînage, on va alterner des périodes en chaînage avant (pour déduire de nouveaux faits de ceux que l'on vient juste de prouver) et des périodes de chaînage arrière (dans lesquelles on cherche de nouveaux buts à prouver).

C'est donc un savant équilibre entre les deux méthodes de recherche, en fonction de règles de recherche. De plus, on peut alterner les phases de recherche en profondeur aux phases de recherche en largeur selon les buts. On dépasse cependant ici le cadre de ce livre.

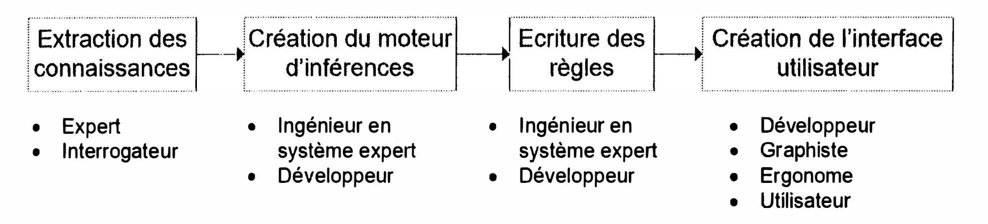
Remarque

Le chaînage mixte est cependant peu utilisé, car il ajoute de la complexité au moteur d'inférences. Il est pourtant beaucoup plus efficace.

5. Étapes de construction d'un système

Pour créer intégralement un système expert, il est important de suivre différentes étapes qui font entrer en jeu des compétences et donc des profils professionnels différents.

Globalement, il y a quatre étapes présentées dans la figure suivante, avec les principaux rôles nécessaires sous chaque étape :



5.1 Extraction des connaissances

La première étape consiste à **extraire les connaissances**. Pour cela, il faut trouver un expert qui sera interrogé pour comprendre les règles sous-jacentes aux décisions qu'il prend dans son travail. Cette phase peut paraître simple mais elle est en fait très complexe. En effet, un expert ne réfléchit pas par règles, il a des automatismes qu'il faut arriver à lui faire expliciter.

Prenons l'exemple des insectes. Face à des insectes peu courants ou inconnus, il paraît assez facile de déterminer des règles qui permettent d'arriver au résultat voulu. Mais quelles règles applique-t-on pour reconnaître une mouche d'un moustique, un escargot d'une limace, une fourmi d'un cloporte, ou encore une coccinelle d'un gendarme ?

C'est donc en posant différentes questions à l'expert que l'on pourra l'amener à déterminer lui-même les règles qu'il applique, souvent inconsciemment. Le travail de l'interrogateur est donc primordial, car ce dernier doit indiquer les zones d'ombre, ou les règles qui ne sont pas assez spécifiques pour faire la discrimination entre deux résultats.

Cette étape peut être très longue, en particulier sur des domaines vastes. De plus, si le domaine d'application possède des risques, il est important de faire vérifier les règles par plusieurs experts, qui pourront les compléter ou les modifier si besoin est.

La réussite du système dépend en très grande partie de cette phase. Il ne faut donc pas la négliger en voulant aller trop vite.

5.2 Création du moteur d'inférences

Si le projet utilise un moteur d'inférences existant, cette phase consistera juste à acquérir celui-ci, voire à le configurer.

Sinon, il faudra **implémenter un moteur d'inférences**, avec les différentes fonctionnalités voulues. Il faudra aussi à ce moment-là créer la structure de la base de faits, et définir les interactions entre le moteur et les bases (de règles et de faits).

Le formalisme des règles sera alors fixé, ce qui pourra avoir un impact important sur les phases suivantes.

5.3 Écriture des règles

La phase suivante consiste à **transformer les différentes règles** obtenues lors de l'extraction des connaissances vers le format voulu par le moteur d'inférences.

À la fin de cette étape, la base de règles doit être complète. Il ne faut pas hésiter à vérifier plusieurs fois la présence de toutes les règles et leur exactitude, car une erreur à ce niveau peut fausser tout le travail fait avec le ou les experts.

Un spécialiste du langage du moteur ainsi qu'en système expert sera donc nécessaire pour cette étape.

5.4 Création de l'interface utilisateur

La dernière étape consiste à **créer l'interface utilisateur**. Nous avons vu précédemment à quel point celle-ci doit être travaillée pour permettre une utilisation simple et juste du moteur d'inférences et des règles.

Dans une première version, on peut imaginer une interface sous la forme d'entrées/sorties dans une console. Cependant, pour une application grand public, une interface graphique sera à préférer. Il est important de faire intervenir au plus tôt les utilisateurs ou au moins leurs représentants, et des spécialistes en ergonomie et graphisme pour définir les différents écrans.

Il existe néanmoins un cas particulier : si le système expert est utilisé par un autre système informatique (et non un humain), il faudra à la place créer les canaux de communication entre les programmes (via des API, des fichiers, des flux, des sockets...).

Une fois le système expert créé, il peut être utilisé.

6. Performance et améliorations

Un point n'a jamais été évoqué jusqu'à présent : celui des performances. Celui-ci est pourtant primordial pour la réussite du système. Nous allons donc voir quels sont les critères de performance et comment construire un système permettant de les améliorer.

6.1 Critères de performance

Les performances d'un système expert, surtout s'il est composé de nombreuses règles, sont primordiales. Le premier critère de performance est le **temps de réponse**. En effet, il faut pouvoir donner une réponse à l'utilisateur dans un temps acceptable.

Ce temps dépend du problème posé. Par exemple, dans notre système expert de géométrie, le temps de réponse sera acceptable s'il reste de l'ordre de la seconde. Et au vu du nombre de règles, il y a peu de risques d'avoir un temps supérieur.

Dans un système expert médical, ou pour aider un garagiste, là encore le temps n'est pas la priorité tant qu'il reste de l'ordre de quelques secondes.

Cependant, si le système expert doit être utilisé dans un environnement dangereux pour prendre une décision (par exemple pour arrêter ou non une machine) ou doit communiquer avec d'autres systèmes, le temps de réponse va devenir un critère primordial pour la réussite du projet.

En plus du temps de réponse, il existe un deuxième critère de performance : l'**utilisation de la mémoire**. En effet, la base de faits va grossir au fur et à mesure de l'application des règles. Dans un moteur à chaînage arrière, le nombre de buts à atteindre peut lui aussi prendre de plus en plus de place. Si le système doit être porté sur un appareil possédant peu de mémoire (comme un robot), il faudra donc bien prendre en compte cet aspect.

Enfin, généralement tous les moyens mis en œuvre pour optimiser le temps de réponse auront un impact négatif sur la mémoire et vice-versa. Il faut donc trouver le bon compromis en fonction des besoins.

6.2 Amélioration des performances par l'écriture des règles

La première façon d'améliorer les performances est de bien travailler sur l'écriture des règles. En effet, il est souvent possible de limiter leur nombre.

Dans notre exemple avec les triangles, on a défini le triangle rectangle isocèle comme étant un triangle rectangle qui est aussi isocèle, mais on aurait pu dire qu'un triangle rectangle isocèle était un triangle possédant un angle droit et deux côtés de même taille. Il est inutile de mettre les deux règles qui, bien que différentes, sont redondantes.

Il faut aussi savoir quels faits seront ou non rentrés par l'utilisateur. Ainsi, on aurait pu définir nos quadrilatères non par leurs côtés parallèles et leurs angles droits, mais par des propriétés sur les diagonales (par exemple qu'elles se coupent en leur milieu pour le parallélogramme, qu'elles sont de la même taille pour le rectangle, ou encore qu'elles se croisent à angle droit pour le losange). Cependant, si l'utilisateur ne possède pas ces informations, ces règles seront inutiles, alors qu'un moteur à chaînage arrière essaiera de prouver ces buts intermédiaires.

L'ordre des règles est lui aussi important. En effet, la plupart des moteurs choisissent la première règle qui correspond à ce qu'ils cherchent, donc la première dont ils possèdent toutes les prémisses en chaînage avant ou la première ayant comme conclusion le but en cours pour le chaînage arrière. Il est donc intéressant de mettre les règles ayant le plus de chances de s'appliquer en premier (ou les plus faciles à prouver ou à réfuter).

Certains moteurs utilisent des critères supplémentaires pour choisir les règles, comme le nombre de prémisses ou la « fraîcheur » des faits utilisés (pour utiliser au maximum les derniers faits obtenus). Il est donc primordial de bien connaître la façon dont le moteur d'inférences agit pour optimiser la base de règles.

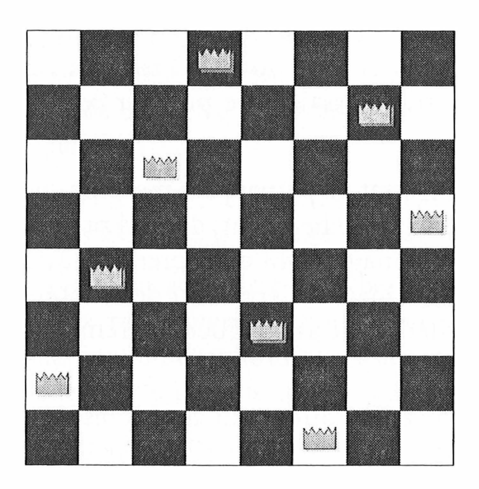
La dernière grande façon d'optimiser cette base est de lui ajouter des index. Ceux-ci ont le même but que les index dans les bases de données : ils permettent de trouver plus rapidement les règles utilisant un fait donné, que ce soit en prémisse (pour les moteurs à chaînage avant) ou en conclusion (dans le cas du chaînage arrière).

6.3 Importance de la représentation du problème

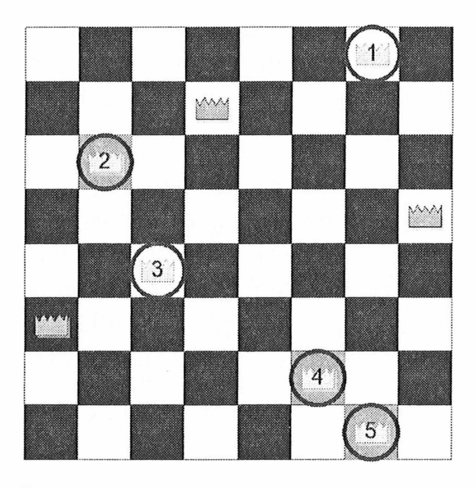
Un moteur d'inférences, avant d'arriver au résultat attendu, va faire de nombreuses tentatives. Il est important de limiter celles-ci pour optimiser les performances du système expert.

Nous allons pour cela nous intéresser à un problème très classique : le problème des « 8 reines ». Le but est de placer sur un échiquier (donc un damier de 8 * 8 cases), huit reines, qui ne doivent pas entrer en conflit, sans s'intéresser aux couleurs des cases. Deux reines sont en conflit si elles sont sur la même ligne ou la même colonne, ou enfin sur la même diagonale.

Les deux figures suivantes indiquent un cas de réussite de placement et un cas incorrect où plusieurs dames entrent en conflit :



Dans cette disposition, il n'y a aucun conflit entre les reines. Il s'agit donc d'une solution possible.



Dans cette disposition par contre, 5 reines sont en conflit entre elles. Les reines 1 et 3 sont sur la même diagonale montante alors que les reines 1 et 5 sont sur la même colonne. 2, 4 et 5 sont sur la même diagonale descendante.

Au total, il y a 92 positions possibles répondant au problème des 8 reines.

Si nous définissons notre problème comme devant nous donner les positions possibles des 8 reines sous la forme (x,y), on voit qu'il y a 64 possibilités par reine (vu qu'il y a 64 cases). Cela nous amène donc à devoir tester 64^8 possibilités (soit plus de 280 billions) ! Un algorithme serait forcément très long à exécuter, même en testant plusieurs milliers de possibilités à la seconde.

On peut prendre en compte le fait que chaque reine doit être sur une case différente des précédentes. Ainsi, pour la première reine, on a 64 possibilités. Lorsque l'on cherche à placer la deuxième reine, une case est déjà prise, il ne reste donc que 63 cases possibles. La troisième pourra être placée sur les 62 cases restantes, et ainsi de suite. Au lieu de 64^8 positions à tester, on n'en a "plus que" 64 * 63 * 62 * 61 * 60 * 59 * 58 * 57. Ce calcul vaut encore plus de 170 billions, mais cela représente déjà un gain appréciable (d'environ 37 %).

Remarque

En mathématiques, on parle d'arrangements. Celui-ci se note A_{64}^{8} *et vaut* $\frac{64!}{(64-8)!}$.

Une réflexion un peu plus poussée peut nous mener à comprendre que les reines doivent être dans des colonnes différentes (sinon elles seraient en conflit comme c'est le cas des reines 1 et 5 dans la figure précédente). On cherche donc pour chaque reine sur quelle ligne (parmi les 8 possibles) elle se situe. Il reste à tester "seulement" 8^8 possibilités, soit un peu moins de 17 millions de possibilités ce coup-ci : la première reine possède 8 cases possibles sur la première colonne, la deuxième 8 cases sur la deuxième colonne...

Enfin, en prenant en compte le fait que les lignes sont, elles aussi, différentes, on voit que si la première reine est placée sur la ligne 1, la deuxième reine n'a plus que 7 lignes possibles pour se placer. On obtient alors 8 * 7 * 6 * 5 * 4 * 3 * 2 * 1 possibilités, que l'on notera 8! (factorielle 8) et qui vaut 40 320. En mathématiques, on parle alors de permutations.

En changeant simplement la représentation du problème, grâce aux contraintes connues, on est donc passé de plus de 200 billions de possibilités à environ 40 000. Si le temps nécessaire pour tester toutes les possibilités dans le premier cas rendait le problème impossible, on voit qu'il sera réalisable et dans un temps acceptable dans le dernier cas.

Le choix de la représentation du problème n'est donc pas anodin, et il est important de bien y réfléchir avant de se lancer dans la création des règles. Les performances peuvent en être fortement impactées.

Remarque

Plus loin dans ce chapitre sera proposée une implémentation pour le système expert sur les polygones et pour les 8 reines en Prolog. C'est la dernière version, utilisant les permutations, qui sera utilisée pour ce problème.

7. Domaines d'application

Le premier système expert est apparu en 1965. Nommé Dendral, il permettait de retrouver les composants d'un matériau à partir d'informations sur sa résonnance magnétique et sur le spectrogramme de masse de celui-ci.

Depuis, les systèmes experts se développent dans de nombreux domaines et sont présents jusque dans les logiciels que l'on utilise tous les jours.

7.1 Aide au diagnostic

Le premier grand domaine d'application des systèmes experts est l'**aide au diagnostic**. En effet, grâce à leur base de règles, ils vont pouvoir tester et éliminer différentes possibilités jusqu'à en trouver une ou plusieurs probables.

On les retrouve ainsi dans les applications médicales pour aider au diagnostic de certaines maladies. On peut citer MYCIN qui permet de déterminer quelle bactérie se trouve dans le corps d'un patient et quel traitement (type d'antibiotique et posologie) lui administrer pour l'aider à guérir ou CADUCEUS, une extension de MYCIN permettant de déterminer plus de 1000 maladies du sang. D'autres applications permettent d'aider le diagnostic de maladies à partir de radiographies ou d'images médicales.

Ils sont très pratiques pour aider à la recherche de pannes en mécanique (comme dans les voitures) ou en électronique (pour les appareils électroménagers), en limitant les pièces potentiellement défectueuses.

On les retrouve aussi tout simplement dans nos ordinateurs lorsqu'une panne ou un défaut est détecté et que le système nous pose différentes questions avant de nous donner une procédure à suivre pour tenter de réparer cette panne (l'assistant de Microsoft Windows en est un bon exemple).

Ils seront enfin de plus en plus utilisés dans les objets connectés que l'on rencontrera, par exemple dans les bracelets permettant de suivre notre santé en temps réel.

7.2 Estimation de risques

Le deuxième grand domaine est l'**estimation des risques**. En effet, à partir des différentes règles de l'expert, le logiciel pourra estimer les risques, de manière à pouvoir y faire face ou tenter de les éviter.

Il existe ainsi des systèmes experts capables de déterminer les risques sur les structures de constructions, pour pouvoir agir rapidement (en renforçant la structure). Les trois plus importants sont Mistral et Damsafe pour les barrages et Kaleidos pour les monuments historiques.

Dans le domaine médical, il existe aussi des systèmes experts pour déterminer les populations à risque, par exemple pour déterminer la probabilité d'une naissance prématurée lors d’une grossesse.

Dans le domaine financier, ils permettent de déterminer les risques pour des crédits ou le montant de l'assurance liée à un prêt. Ils sont utilisés enfin en détection des fraudes en retrouvant les transactions qui paraissent anormales (par exemple pour déterminer si une carte bleue a été volée).

L'estimation de risques ne se limite pas à ces domaines et ils peuvent se retrouver partout. "Rice-crop doctor" est un système expert qui permet d'indiquer les risques de maladies des pieds de riz, et est donc utilisé en agriculture.

7.3 Planification et logistique

Les systèmes experts permettant de tester plusieurs possibilités en suivant des règles sont très adaptés à la création de **plannings** devant respecter différentes contraintes imposées. Ce problème, surtout lorsqu'il est de taille moyenne ou grande, est impossible à résoudre par un seul être humain dans un temps raisonnable.

On les retrouve ainsi dans tous les domaines nécessitant de la planification, que ce soit dans les écoles pour les emplois du temps, les aéroports pour les vols des avions ou les hôpitaux pour les plannings d'utilisation des salles d'opération.

En logistique, ils permettent aussi d'optimiser le rangement dans les entrepôts ou les tournées de livraison.

7.4 Transfert de compétences et connaissances

Les systèmes experts manipulent des connaissances, et il est intéressant de les utiliser pour le **transfert de compétences**.

Ils peuvent ainsi être utilisés dans l'éducation et la formation : ils permettent d'indiquer à l'apprenant les étapes qui vont l'aider à déterminer un fait ou les règles qui s'appliquent à un domaine. Ils peuvent le guider dans son raisonnement, et lui apprendre certains "réflexes", acquis par des experts grâce à des années d'expérience.

L'identification d'objets, de plantes, d'animaux ou de pierres est aisée via un système expert : grâce à des questions, ce dernier va pouvoir indiquer ce que l'on cherche à identifier. L'importance est alors mise sur l'ordre des questions. Cet enchaînement de questions est appelé une "clé de détermination". Il en existe de nombreuses, pour tous les domaines, toujours basées sur un ensemble de règles.

Enfin, non seulement ils peuvent utiliser les connaissances d'un expert, mais ils peuvent aussi manipuler ces connaissances pour en faire de nouveaux faits, inconnus jusqu'alors. La démonstration automatique consiste donc à donner au système des faits (mathématiques par exemple) et des règles indiquant comment les combiner. On laisse alors le logiciel trouver de nouvelles démonstrations pour des théorèmes connus, ou chercher de nouveaux théorèmes.

7.5 Autres applications

Les applications des systèmes experts ne se limitent pas aux grands domaines précédents. Il est ainsi possible de les trouver dans des domaines beaucoup plus spécifiques.

Les outils actuels de traitement de texte proposent tous des correcteurs orthographiques et grammaticaux. Si l'orthographe se base seulement sur la connaissance ou non des mots tapés, la grammaire est beaucoup plus complexe et demande un système expert contenant les différentes règles (par exemple l'accord entre le nom et l'adjectif). Les faits sont les mots utilisés, avec leur genre et leur nombre, ainsi que les structures de phrases. Ces systèmes sont de plus en plus performants.

On peut aussi trouver des systèmes experts pour aider à la configuration de machines ou de systèmes en fonction de faits sur leur utilisation, sur le système sous-jacent ou sur le service sur lequel il est installé. D'autres permettent de contrôler en temps réel des systèmes, on les retrouve même à la NASA pour certaines opérations sur les navettes spatiales.

Enfin, on trouve des systèmes experts dans les logiciels de création de pièces mécaniques. Ces systèmes permettent ainsi de respecter des règles de design qu'il est difficile de prendre en compte pour un opérateur humain, de par leur complexité ou leur nombre.

Les systèmes experts peuvent donc se retrouver dans tous les domaines où un expert humain est utile actuellement, ce qui leur donne une grande capacité d'adaptation et de nombreuses applications.

8. Création d'un système expert en C#

Coder un moteur d'inférences générique et complet en C# est une tâche complexe à faire. De plus, il existe des moteurs disponibles (gratuitement ou non) qu'il est facile d'adapter à son problème.

Nous allons cependant nous intéresser à la création du système expert permettant de donner le nom d'un polygone à partir de ses caractéristiques.

Ce système expert sera cependant adaptable à d'autres problèmes proches. De plus, le code C# est compatible avec Windows Phone 8, Windows 8, Silverlight 5 et le framework .NET 4 ainsi que les versions supérieures. Le programme principal est un programme de type console pour Windows.

8.1 Détermination des besoins

Ce système doit être adaptable à de nombreux problèmes équivalents, de type identification, à partir d'informations entrées par l'utilisateur.

Ici, le système part des données fournies par l'utilisateur pour essayer de déterminer la forme qu'il cherche à reconnaître. Nous n'avons pas de but précis. Un moteur à chaînage avant est donc à préférer pour ce problème, en plus d'être plus simple d'implémentation.

De plus, d'après nos règles, nous aurons deux types de faits :

- Des faits dont les valeurs seront entières, comme l'ordre du polygone, ou le nombre de côtés de même taille.
- Des faits dont les valeurs seront des booléens, comme la présence ou non d'un angle droit, ou le fait d'être un triangle.

Notre système expert devra donc prendre en compte ces deux types de faits.

L'utilisateur doit aussi pouvoir facilement utiliser le système expert. Les règles devront donc être écrites dans un langage proche du langage naturel, et l'interface devra être gérée séparément du moteur pour le rendre plus générique (ici on ne fera qu'une interface en mode console).

Enfin, il serait aussi intéressant de connaître le dernier nom trouvé pour la forme et les noms intermédiaires qui ont été utilisés (par exemple, on a utilisé "Triangle", puis "Triangle rectangle" et "Triangle isocèle" et enfin on a fini sur "Triangle rectangle isocèle"). Nous chercherons donc à implémenter un moyen de connaître l'ordre des faits.

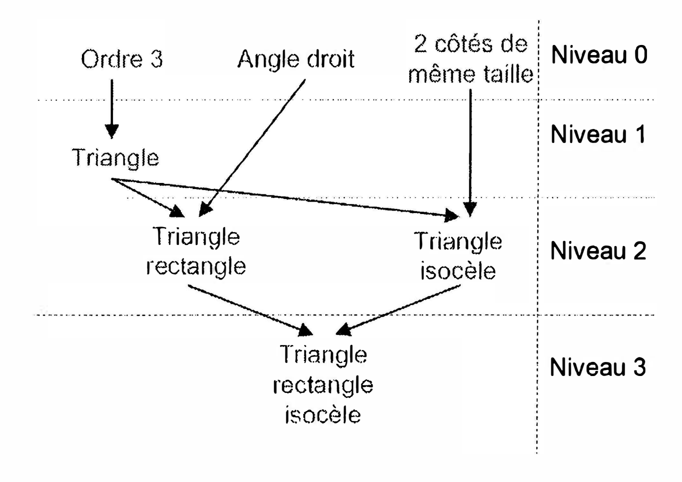
8.2 Implémentation des faits

Les faits sont les premiers à être codés. Comme nous avons deux types de faits, une interface générique est codée, qui est ensuite implémentée dans les deux types particuliers.

Un fait doit posséder :

- Un nom, qui est une chaîne de caractères.
- Une valeur, qui est de type `Object` au niveau de l'interface (dans les classes concrètes, il s'agira d'un entier ou d'un booléen, mais on pourrait imaginer d'autres types de faits).
- Un niveau, qui correspond à sa place dans l'arbre des décisions : le niveau sera de 0 pour les faits donnés par l'utilisateur, et on augmentera de 1 le niveau pour les faits inférés. Ce niveau sera donc modifiable par le moteur.
- Une question à poser à l'utilisateur, pour les faits qui peuvent être demandés (les faits ne pouvant être qu'inférés n'auront pas de question).

Dans l'exemple du triangle rectangle isocèle, voici dans la figure suivante les niveaux associés à chaque conclusion obtenue. Ce niveau s'incrémente de 1 par rapport aux faits utilisés en prémisses. Cela nous permet de savoir quel est le fait de plus haut niveau (celui qui est sûrement le plus important pour l'utilisateur) :



Une interface **IFact** est créée qui possède plusieurs méthodes permettant de lire les attributs, et une méthode pour modifier le niveau du fait :

```
using System;
public interface IFact
{
    String Name();
    Object Value();
    int Level();
    String Question();

    void SetLevel(int p);
}
```

Deux classes vont implémenter cette interface : `IntFact` qui sera un fait à valeur entière et `BoolFact`, un fait à valeur booléenne. Ces classes vont posséder des attributs protégés, accessibles via les méthodes de l'interface.

Un constructeur initialisant les différentes valeurs avec des valeurs par défaut pour la question et le niveau est ajouté. La méthode `ToString` pour l'affichage créera une chaîne du type "Ordre=3 (0)" signifiant que le fait "Ordre" vaut 3, et qu'il a été donné par l'utilisateur (niveau 0).

Voici donc le code de la classe **IntFact** :

```
using System;
internal class IntFact : Ifact
{
    protected String name;
    public String Name()
    {
        return name;
    }

    protected int value;
    public object Value()
    {
        return value;
    }

    protected int level;
    public int Level()
    {
        return level;
```

```
    }
    public void SetLevel(int l)
    {
        level = l;
    }

    protected String question = null;
    public String Question()
    {
        return question;
    }

    public IntFact(String _name, int _value, String _question =
null, int _level = 0)
    {
        name = _name;
        value = _value;
        question = _question;
        level = _level;
    }

    public override String ToString()
    {
        return name + "=" + value +" (" + level + ")";
    }
}
```

Il en est de même pour la classe **BoolFact** représentant un fait booléen. Pour l'affichage, si le fait est vrai on le représentera seulement sous la forme Triangle(1) (indiquant que le fait "Triangle" a été obtenu au niveau 1, donc à partir des faits entrés par l'utilisateur), et !Triangle(1) si le fait est faux.

Voici son code :

```
using System;
internal class BoolFact : Ifact
{
    protected String name;
    public String Name()
    {
        return name;
    }

    protected bool value;
```

```
    public object Value()
    {
        return value;
    }

    protected int level;
    public int Level()
    {
        return level;
    }
    public void SetLevel(int l)
    {
        level = l;
    }

    protected String question = null;
    public String Question()
    {
        return question;
    }

    public BoolFact(String _name, bool _value, String _question =
null, int _level = 0)
    {
        name = _name;
        value = _value;
        question = _question;
        level = _level;
    }

    public override String ToString()
    {
        String res = "";
        if (!value)
        {
            res += "!";
        }
        res += name + " (" + level + ")";
        return res;
    }
}
```

8.3 Base de faits

Une fois les faits définis, la base de faits, nommée **FactsBase**, peut être implémentée. Celle-ci, vide à l'origine, sera remplie petit à petit par le moteur. Elle contient donc une liste de faits, à initialiser dans le constructeur et accessible en lecture via une propriété :

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

internal class FactsBase
{
    protected List<IFact> facts;
    public List<IFact> Facts
    {
            get
            {
                  return facts;
            }
    }

    public FactsBase()
    {
            facts = new List<IFact>();
    }
}
```

Deux autres méthodes sont implémentées : l'une pour ajouter un fait en base et l'autre pour vider complètement celle-ci (pour pouvoir traiter un nouveau cas par exemple) :

```
    public void Clear()
    {
            facts.Clear();
    }

    public void AddFact(IFact f)
    {
            facts.Add(f);
    }
```

De plus, il faut aussi deux méthodes plus spécifiques :

- Une méthode `Search` permettant de rechercher un fait dans la base, elle prend donc un nom en paramètre et renvoie le fait si elle le trouve (ou `null` sinon).
- Une méthode `Value` permettant de retrouver la valeur d'un fait dans la base, toujours à partir de son nom donné en paramètre. Si le fait n'existe pas, cette méthode renvoie `null`. Pour être générique, la valeur renvoyée est de type **`Object`**.

Voici leur code, qui utilise Linq pour faire les recherches dans les listes.

```
public IFact Search(String _name)
{
        return facts.FirstOrDefault(x => x.Name().Equals(_name));
}

public Object Value(String _name)
{
        IFact f = facts.FirstOrDefault(x => x.Name().Equals(_name));
        if (f != null)
        {
                return f.Value();
        }
        else
        {
                return null;
        }
}
```

La base de faits est alors complète.

8.4 Règles et base de règles

Après les faits, il est possible de coder les règles. Celles-ci contiennent trois propriétés :

- Un nom, sous la forme d'une chaîne.
- Une liste de faits formant les prémisses de la règle (la partie gauche).
- Un fait qui est la conclusion de la règle (la partie droite).

Le code de base de la classe **Rule** est donc le suivant :

```
using System;
using System.Collections.Generic;

public class Rule
{
    public List<IFact> Premises { get; set; }
    public IFact Conclusion { get; set; }
    public String Name { get; set; }

    public Rule(String _name, List<IFact> _premises, IFact _conclusion)
    {
        Name = _name;
        Premises = _premises;
        Conclusion = _conclusion;
    }
}
```

Pour des raisons de lisibilité si besoin, une méthode `ToString` est ajoutée. Elle écrit la règle sous la forme :

```
Nom : IF (premisse1 AND premisse2 AND ...) THEN conclusion
```

Le code de cette méthode utilise la méthode `String.Join()` permettant de transformer un tableau en une chaîne de caractères, avec le séparateur indiqué (ici "AND").

```
   public override string ToString()
   {
          return Name + " : IF (" + String.Join(" AND ",
Premises)+ ") THEN " + Conclusion.ToString();
   }
```

La base de règles est nommée **RulesBase**. Celle-ci contient une liste de règles, accessible en lecture ou écriture via une propriété, et un constructeur qui initialise la liste.

```
using System.Collections.Generic;

internal class RulesBase
{
    protected List<Rule> rules;
    public List<Rule> Rules
    {
        get
        {
```

```
            return rules;
        }
        set
        {
            rules = value;
        }
    }

    public RulesBase() {
        rules = new List<Rule>();
    }
}
```

Plusieurs méthodes sont présentes pour ajouter une règle, en enlever une ou vider toute la base de règles :

```
    public void ClearBase()
    {
        rules.Clear();
    }

    public void AddRule(Rule r)
    {
        rules.Add(r);
    }

    public void Remove(Rule r)
    {
        rules.Remove(r);
    }
```

Les deux bases sont créées ainsi que les structures contenues dans celles-ci.

8.5 Interface

Le dernier élément gravitant autour du moteur dans un système expert est l'interface utilisateur (ou IHM). Il est nécessaire de commencer par définir une interface indiquant les méthodes que devront implémenter toutes les IHM.

Nous avons besoin de deux méthodes permettant de demander à l'utilisateur la valeur d'un fait, entier (`AskIntValue`) ou booléen (`AskBoolValue`). De plus, il faut deux méthodes pour afficher les faits (`PrintFacts`) ou les règles (`PrintRules`).

Voici le code de l'interface **HumanInterface** :

```
using System;
using System.Collections.Generic;

public interface HumanInterface
{
    int AskIntValue(String question);
    bool AskBoolValue(String question);
    void PrintFacts(List<IFact> facts);
    void PrintRules(List<Rule> rules);
}
```

Ces méthodes sont volontairement très génériques : selon le programme voulu, les entrées pourront se faire en ligne de commande, via un formulaire web, dans un champ de texte, par un slider... De la même façon, l'affichage des règles et des faits reste libre (texte, tableau, liste à puce, graphique...).

Dans notre cas, le programme principal **Program** implémente ces méthodes, en utilisant uniquement la console pour les entrées/sorties. Pour la lecture des valeurs, une conversion pour les entiers et une lecture yes/no pour les booléens sont présentes (il n'y a pas de gestion d'erreurs).

```
using System;
using System.Collections.Generic;
using System.Linq;

class Program : HumanInterface
{
    static void Main(string[] args)
    {
        // À remplir plus tard
    }

    public int AskIntValue(String p)
    {
        Console.Out.WriteLine(p);
        try {
            return int.Parse(Console.In.ReadLine());
        }
        catch (Exception)
        {
            return 0;
        }
    }
```

```
    public bool AskBoolValue(String p)
    {
        Console.Out.WriteLine(p + " (yes, no)");
        String res = Console.In.ReadLine();
        if (res.Equals("yes"))
        {
            return true;
        }
        else
        {
            return false;
        }
    }
}
```

Pour les affichages, il s'agit de simples appels aux méthodes `ToString()`. Cependant, pour l'affichage des faits, nous utilisons Linq pour n'afficher que les faits dont l'ordre est supérieur à 0 (donc uniquement les faits inférés) et par ordre décroissant de niveau. De cette façon, les faits obtenus en dernier et de plus haut niveau seront affichés en premier : "Triangle rectangle" sera ainsi affiché avant "Triangle".

```
    public void PrintFacts(List<IFact> facts)
    {
        String res = "Solution(s) trouvée(s) : \n";
        foreach (IFact f in facts.Where(x => x.Level() >
0).OrderByDescending(x => x.Level()))
        {
            res += f.ToString() + "\n";
        }
        Console.Out.Write(res);
    }

    public void PrintRules(List<Rule> rules)
    {
        String res = "";
        foreach (Rule r in rules)
        {
            res += r.ToString() + "\n";
        }
        Console.Out.Write(res);
    }

}
```

Cette classe sera complétée ultérieurement.

8.6 Moteur d'inférences

Il faut maintenant implémenter la pièce centrale du système : un petit moteur d'inférences à chaînage avant, facilement adaptable à d'autres problèmes d'identification (de plantes, d'insectes, d'animaux, de roches...).

Cette classe, nommée **Motor**, contient tout d'abord trois attributs : une base de faits, une base de règles et une interface. Le constructeur va initialiser les deux bases, et récupérer l'IHM passée en paramètre.

La base de cette classe est donc la suivante :

```
using System;
using System.Collections.Generic;

public class Motor
{
    private FactsBase fDB;
    private RulesBase rDB;
    private HumanInterface ihm;

    public Motor(HumanInterface _ihm)
    {
        ihm = _ihm;
        fDB = new FactsBase();
        rDB = new RulesBase();
    }
}
```

Avant de continuer, il faut implémenter une méthode permettant de demander à l'utilisateur la valeur d'un fait et le créer (pour l'ajouter ultérieurement à la base de faits). Comme le fait peut être de différents types (`IntFact` ou `BoolFact`) et de façon à rester le plus générique possible, une classe statique servira de fabrique de faits. C'est elle qui se chargera de créer le fait du bon type et elle nous renverra un `IFact`.

Le code de cette classe **FactFactory** est le suivant (il sera complété ultérieurement) :

```
internal static class FactFactory
```

```
{
    internal static IFact Fact(IFact f, Motor m)
    {
        IFact newFact;
        if (f.GetType().Equals(typeof(IntFact)))
        {
            // C'est un fait à valeur entière
            int value = m.AskIntValue(f.Question());
            newFact = new IntFact(f.Name(), value, null, 0);
        }
        else
        {
            // C'est un fait à valeur booléenne
            bool value = m.AskBoolValue(f.Question());
            newFact = new BoolFact(f.Name(), value, null, 0);
        }
        return newFact;
    }
}
```

La classe **Motor** peut alors être complétée. Les deux méthodes appelées par `FactFactory`, à savoir `AskIntValue` et `AskBoolValue`, sont ajoutées. Elles ne seront que des redirections vers les méthodes de l'interface.

```
internal int AskIntValue(string p)
{
    return ihm.AskIntValue(p);
}

internal bool AskBoolValue(string p)
{
    return ihm.AskBoolValue(p);
}
```

C'est ensuite au tour d'une méthode `CanApply`, qui indique si une règle peut s'appliquer (c'est-à-dire si toutes ses prémisses sont vérifiées). Cette méthode doit donc parcourir les faits en prémisses et vérifier s'ils existent en base de faits. Plusieurs cas peuvent se présenter :

– Le fait n'est pas présent en base de faits : soit il possède une question, et dans ce cas il faut demander la valeur à l'utilisateur et l'ajouter en base de faits, soit il ne possède pas de question (c'est donc un fait uniquement inférable), et la règle ne pourra pas s'appliquer.

– Le fait est présent en base de fait : soit la valeur correspond, et dans ce cas, on passe au fait suivant, soit la valeur ne correspond pas, et alors, la règle ne s'appliquera pas.

De plus, pendant le parcours des faits, il faut chercher quel est le niveau maximum des prémisses. En effet, si la règle s'applique, le fait inféré aura pour niveau celui des prémisses + 1. Ainsi, une règle ayant des prémisses de niveau 0 et 3 créera un fait de niveau 4. C'est ce dernier qui sera renvoyé. Comme il est obligatoirement positif, on renverra -1 dans le cas où la règle n'est pas applicable.

```
private int CanApply(Rule r)
{
    int maxlevel = -1;
    // On vérifie si chaque prémisse est vraie
    foreach (IFact f in r.Premises)
    {
        IFact foundFact = fDB.Search(f.Name());
        if (foundFact == null) {
            // Ce fait n'existe pas dans la base actuellement
            if (f.Question() != null) {
                // On le demande à l'utilisateur
                // et on l'ajoute en base
                foundFact = FactFactory.Fact(f, this);
                fDB.AddFact(foundFact);
                maxlevel = Math.Max(maxlevel, 0);
            }
            else {
                // On sait que la règle ne s'applique pas
                return -1;
            }
        }

        // On a un fait en base, on vérifie sa valeur
        if (!foundFact.Value().Equals(f.Value()))
        {
            // Elle ne correspond pas
            return -1;
        }
        else
        {
            // Elle correspond
            maxlevel = Math.Max(maxlevel, foundFact.Level());
```

```
            }
        }

        return maxlevel;
    }
```

La méthode suivante, `FindUsableRule`, va permettre de chercher, parmi toutes les règles de la base, la première qui pourrait s'appliquer. Elle s'appuie donc sur `CanApply`.

Si une règle peut s'appliquer, un tuple contenant deux paramètres est renvoyé : la règle que l'on peut appliquer, et le niveau maximum des prémisses. Dans le cas où aucune règle ne peut s'appliquer, on renvoie `null`.

```
    private Tuple<Rule, int> FindUsableRule(RulesBase rBase) {
        foreach(Rule r in rBase.Rules) {
            int level = CanApply(r);
            if (level != -1) {
                return Tuple.Create(r, level);
            }
        }
        return null;
    }
```

La méthode principale du moteur, `Solve()`, permet de résoudre complètement un problème. L'algorithme de cette méthode est assez simple :

- Une copie locale de toutes les règles existantes est faite et on initialise la base de faits.
- Tant qu'on peut appliquer une règle :
  - Elle est appliquée et le fait inféré est ajouté à la base de faits (en incrémentant son niveau).
  - Elle est supprimée (pour ne pas la réexécuter plus tard sur le même problème).
- Quand il n'y a plus de règles, on affiche les résultats.

Voici le code de cette méthode :

```
    public void Solve()
    {
        // On fait une copie des règles existantes
        // + création d'une base de faits vierge
```

```
        bool moreRules = true;
        RulesBase usableRules = new RulesBase();
        usableRules.Rules = new List<Rule>(rDB.Rules);
        fDB.Clear();

        // Tant qu'il existe des règles à appliquer
        while (moreRules)
        {
            // Cherche une règle à appeler
            Tuple<Rule, int> t = FindUsableRule(usableRules);

            if (t!= null)
            {
                // Applique la règle et ajoute le nouveau fait
                // à la base
                IFact newFact = t.Item1.Conclusion;
                newFact.SetLevel(t.Item2 + 1);
                fDB.AddFact(newFact);

                // Enlève la règle des règles applicables
                usableRules.Remove(t.Item1);
            }
            else
            {
                // Plus de règles possibles : on s'arrête
                moreRules = false;
            }
        }

        // Écriture du résultat
        ihm.PrintFacts(fDB.Facts);
    }
```

À ce stade, l'application des règles est complètement gérée. Il ne manque plus qu'un moyen d'ajouter des règles. Comme pour demander à l'utilisateur un fait, il va falloir un moyen pour lire la règle écrite, en extraire les faits et créer un fait de la bonne classe selon qu'il s'agisse d'un fait entier ou booléen.

Pour simplifier le travail, il faut donc ajouter une deuxième méthode dans **FactFactory**, qui permettra de créer un fait à partir d'une chaîne. Les faits seront exprimés sous la forme "Nom=Valeur (question)" pour un fait entier ou "Nom" / "!Nom" pour un fait booléen.

Voici le code de cette méthode `Fact`, qui renvoie un `IFact` en fonction de la chaîne reçue. Le code consiste à découper la chaîne selon les séparateurs ('=', '(', ')', '!'), à enlever les espaces en début et fin de chaîne (méthode `Trim()`) et à créer le bon fait avec la bonne valeur. On remplit aussi la question si elle est fournie (et donc si le fait n'est pas seulement inféré).

```
    internal static IFact Fact(string factStr)
    {
        factStr = factStr.Trim();
        if (factStr.Contains("="))
        {
            // Il y a un symbole '=' donc c'est un IntFact,
            // on sépare le nom de la valeur
            String[] nameValue = factStr.Split(new String[] {
"=", "(", ")" }, StringSplitOptions.RemoveEmptyEntries);
            if (nameValue.Length >= 2)
            {
                String question = null;
                if (nameValue.Length == 3)
                {
                    // On peut le demander, donc on récupère
                    // la question liée
                    question = nameValue[2].Trim();
                }
                return new IntFact(nameValue[0].Trim(),
int.Parse(nameValue[1].Trim()), question);
            }
            else
            {
                // Syntaxe incorrecte
                return null;
            }
        }
        else
        {
            // Pas d'égalité, c'est un fait de classe BoolFact
            bool value = true;
            if (factStr.StartsWith("!"))
            {
                value = false;
                factStr = factStr.Substring(1).Trim();
                // On enlève le ! du nom
            }
```

```
            String[] nameQuestion = factStr.Split(new String[] {
"(", ")" }, StringSplitOptions.RemoveEmptyEntries);
            String question = null;
            if (nameQuestion.Length == 2)
            {
                // On récupère la question si on peut
                question = nameQuestion[1].Trim();
            }
            return new BoolFact(nameQuestion[0].Trim(), value,
question);
        }
    }
```

La classe **Motor** peut alors être terminée avec une méthode qui permettra d'ajouter une règle à partir d'une chaîne de caractères (ce qui est plus simple pour l'utilisateur). Cette méthode commence par couper la chaîne au niveau du symbole ':' pour séparer le nom de la règle. Ensuite, en séparant la chaîne au niveau des mots-clés 'IF' et 'THEN', on peut séparer les prémisses de la conclusion. Pour les prémisses, on les sépare enfin par la présence de 'AND'.

Remarque

Les règles fournies, de par l'implémentation de cette méthode, ne pourront pas contenir les mots IF, THEN ni les symboles suivants : '=', ':', '(', ')'. En effet, ceux-ci servent de séparateurs.

Voici donc le code de cette méthode `AddRule` :

```
    public void AddRule(string ruleStr)
    {
        // Séparation nom : règle
        String[] splitName = ruleStr.Split(new String[] {" : "},
StringSplitOptions.RemoveEmptyEntries);
        if (splitName.Length == 2)
        {
            String name = splitName[0];
            // Séparation premisses THEN conclusion
            String[] splitPremConcl = splitName[1].Split(new String[]
{"IF ", " THEN " }, StringSplitOptions.RemoveEmptyEntries);
            if (splitPremConcl.Length == 2)
            {
                // Lecture des premisses
                List<IFact> premises = new List<IFact>();
                String[] premisesStr =
splitPremConcl[0].Split(new String[] {" AND "},
```

```
StringSplitOptions.RemoveEmptyEntries);
                foreach (String prem in premisesStr)
                {
                    IFact premise = FactFactory.Fact(prem);
                    premises.Add(premise);
                }

                // Lecture de la conclusion
                String conclusionStr = splitPremConcl[1].Trim();
                IFact conclusion =
FactFactory.Fact(conclusionStr);

                // Création de la règle et ajout
                rDB.AddRule(new Rule(name, premises, conclusion));
            }
        }
    }
```

8.7 Saisie des règles et utilisation

Le système expert est complet. Il est générique, et peut donc actuellement s'appliquer à n'importe quel problème.

La classe `Program` est complétée par la méthode `main` pour pouvoir résoudre le problème de noms de polygones.

Le `main` sera simple, il ne fera qu'appeler la méthode "`Run`" décrite juste après :

```
static void Main(string[] args)
{
        Program p = new Program();
        p.Run();
}
```

La méthode `Run` va devoir :

- Créer un nouveau moteur.
- Ajouter les règles, via leur version textuelle. Ici, seules les onze règles pour les triangles et les quadrilatères seront ajoutées, mais il est facile de les compléter par de nouvelles règles. Elles sont aussi faciles à lire pour l'utilisateur ou le concepteur de celles-ci.
- Lancer la résolution du problème.

Voici son code :

```
    public void Run()
    {
        // Moteur
        Console.Out.WriteLine("** Création du moteur **");
        Motor m = new Motor(this);

        // Règles
        Console.Out.WriteLine("** Ajout des règles **");
        m.AddRule("R1 : IF (Ordre=3(Quel est l'ordre ?)) THEN
Triangle");
        m.AddRule("R2 : IF (Triangle AND Angle Droit(La figure a-
t-elle au moins un angle droit ?)) THEN Triangle Rectangle");
        m.AddRule("R3 : IF (Triangle AND Cotes Egaux=2(Combien la
figure a-t-elle de côtés égaux ?)) THEN Triangle Isocèle");
        m.AddRule("R4 : IF (Triangle Rectangle AND Triangle
Isocèle) THEN Triangle Rectangle Isocèle");
        m.AddRule("R5 : IF (Triangle AND Cotes Egaux=3(Combien la
figure a-t-elle de côtés égaux ?)) THEN Triangle Equilateral");
        m.AddRule("R6 : IF (Ordre=4(Quel est l'ordre ?)) THEN
Quadrilatère");
        m.AddRule("R7 : IF (Quadrilatère AND Cotes
Paralleles=2(Combien y a-t-il de côtés parallèles entre eux - 0,
2 ou 4)) THEN Trapeze");
        m.AddRule("R8 : IF (Quadrilatère AND Cotes
Paralleles=4(Combien y a-t-il de côtés parallèles entre eux - 0,
2 ou 4)) THEN Parallélogramme");
        m.AddRule("R9 : IF (Parallélogramme AND Angle Droit(La
figure a-t-elle au moins un angle droit ?)) THEN Rectangle");
        m.AddRule("R10 : IF (Parallélogramme AND Cotes
Egaux=4(Combien la figure a-t-elle de côtés égaux ?)) THEN
Losange");
        m.AddRule("R11 : IF (Rectangle AND Losange THEN Carré");

        // Résolution
        while (true)
        {
            Console.Out.WriteLine("\n** Résolution **");
            m.Solve();
        }
    }
```

Et voici le type de sortie que l'on peut obtenir (ici on détermine un triangle rectangle isocèle puis un rectangle) :

```
** Création du moteur **
** Ajout des règles **

** Résolution **
Quel est l'ordre ?
3
La figure a-t-elle au moins un angle droit ? (yes, no)
yes
Combien la figure a-t-elle de côtés égaux ?
2
Solution(s) trouvée(s) :
Triangle Rectangle Isocèle (3)
Triangle Rectangle (2)
Triangle Isocèle (2)
Triangle (1)

** Résolution **
Quel est l'ordre ?
4
Combien y a-t-il de côtés parallèles entre eux - 0, 2 ou 4
4
La figure a-t-elle au moins un angle droit ? (yes, no)
yes
Combien la figure a-t-elle de côtés égaux ?
2
Solution(s) trouvée(s) :
Rectangle (3)
Parallélogramme (2)
Quadrilatère (1)
```

9. Utilisation de Prolog

Coder un système expert en programmation objet se fait, mais ce n'est pas le paradigme de programmation le plus adapté. Les langages en **programmation logique** sont faits pour exécuter ce type de tâche. L'écriture de code sera alors plus simple, toute la logique du moteur étant déjà implémentée dans le cœur du langage.

9.1 Présentation du langage

Prolog, pour PROgrammation LOGique est un des premiers langages de ce paradigme, créé en 1972 par deux Français, Alain Colmerauer et Philippe Roussel.

Ce n'est cependant pas le seul langage en programmation logique. On peut aussi citer Oz ou CLIPS. Ces langages se rapprochent de ceux de programmation fonctionnelle (LISP, Haskell...) qui peuvent aussi être utilisés pour des systèmes experts.

Prolog contient un **moteur d'inférences à chaînage arrière**, avec backtracking. On lui donne un but (qui doit être un fait) qu'il va essayer de résoudre. Si ce but contient une variable, alors il cherchera toutes les valeurs possibles pour celle-ci. Si le but contient un attribut, alors il confirmera ou infirmera le fait.

Prolog fonctionne sur la base de **prédicats**. Chaque prédicat peut être soit un fait avéré, soit un fait inféré grâce à des règles. Par exemple ordre(3) est un prédicat avec un paramètre qui est un fait donné par l'utilisateur. Par contre, nom(triangle) sera un fait inféré par la règle "SI ordre(3) ALORS nom(triangle)".

Le langage contient aussi des prédicats fournis, permettant par exemple d'écrire et lire dans la console de Prolog, de charger un fichier, ou de manipuler des listes.

Dans la suite, nous utiliserons le SWI-Prolog, téléchargeable gratuitement sur le site http://www.swi-prolog.org/ et disponible pour Windows, Mac OS X ou Linux.

9.2 Syntaxe du langage

Remarque

Seule la syntaxe nécessaire à la compréhension des exemples est ici fournie. N'hésitez pas à lire la documentation de votre implémentation de Prolog pour en savoir plus. De plus, selon la version utilisée, la syntaxe peut légèrement différer. En annexe se trouve une explication de l'utilisation de SWI-Prolog pour Windows.

9.2.1 Généralités

En Prolog, il faut différencier deux parties :

- Le fichier contenant les règles et les faits (donc l'ensemble des prédicats définis pour le problème).
- La console qui sert uniquement à l'interaction avec l'utilisateur.

Dans le fichier chargé, on peut tout d'abord trouver des **commentaires**. Ceux-ci sont de deux types :

```
% Commentaire sur une seule ligne
```

```
/* Commentaire pouvant
prendre plusieurs lignes */
```

9.2.2 Prédicats

On trouve ensuite les **prédicats** qui doivent forcément commencer par des minuscules. Ils se terminent toujours par un point, comparable au point-virgule annonçant la fin d'une instruction dans un langage objet. Les prédicats peuvent avoir des paramètres, qui sont entre parenthèses. Si ces paramètres sont des **variables**, alors leur nom doit commencer par une majuscule, sinon ils commencent par une minuscule (il existe un cas particulier : '_' représente une variable anonyme, dont la valeur n'a pas d'importance).

Voici des exemples de prédicats :

```
manger(chat, souris).
manger(souris, fromage).

fourrure(chat).
fourrure(souris).
```

On peut ainsi y lire que le chat mange la souris, et que la souris mange le fromage. De plus, le chat et la souris possèdent de la fourrure.

9.2.3 Poser des questions

Ce simple fichier est déjà exécutable. Il suffit de le charger dans la console. On peut alors poser des questions. Lorsque Prolog donne un résultat, si celui-ci ne se termine pas par un point, cela signifie qu'il y a potentiellement d'autres réponses. L'appui sur la touche ';' permet d'obtenir les suivantes, jusqu'à l'échec (`false`.) ou la présence d'un point.

Le premier type de question est simplement de savoir si un fait est vrai ou faux. On peut ainsi demander si le chat mange du fromage (c'est faux) ou si la souris mange du fromage (c'est vrai). La présence d'un point à la fin de la réponse indique que Prolog ne pourra fournir d'autres réponses :

```
?- manger(chat, fromage).
false.

?- manger(souris, fromage).
true.
```

Remarque

Les lignes commençant par "?-" sont toujours tapées dans la console de Prolog. "?-" en est le prompt.

On peut aussi lui demander qui mange du fromage. Ce coup-ci, on utilise une variable, dont le nom doit commencer par une majuscule. Par convention, on utilise X, mais ce n'est pas une obligation (on pourrait l'appeler MangeursDeFromage si on le souhaitait) :

```
?- manger(X, fromage).
X = souris.
```

Prolog a alors fait ce que l'on appelle une **unification** : il a cherché les valeurs possibles pour la variable X. Ici, il n'y a qu'un seul résultat.

De même, on peut lui demander qui mange qui/quoi, et qui a de la fourrure. Ce coup-ci, on va demander les résultats suivants avec ';' :

```
?- manger(X, Y).
X = chat,
Y = souris ;
X = souris,
Y = fromage.

?- fourrure(X).
X = chat ;
X = souris.
```

Dans les deux cas, on voit que l'on obtient deux résultats (les couples (chat, souris) et (souris, fromage) pour le premier cas, et chat ainsi que souris pour le deuxième).

9.2.4 Écriture des règles

Notre fichier chargé ne contient que quatre lignes et ce ne sont que des faits, mais on a déjà un programme fonctionnel. Cependant, surtout dans le cas d'un système expert, il faut pouvoir entrer de nouvelles règles.

Le format des **règles** est le suivant :

```
conclusion :-
     premisse1,
     premisse2,
     % autres prémisses
     premisseN.
```

Là encore, des variables peuvent être unifiées. Soit une règle disant que les ennemis de nos ennemis sont nos amis (en utilisant la relation manger) qui s'écrit donc :

```
amis(X, Y) :-
     manger(X, Z),
     manger(Z, Y).
```

Une fois cette règle ajoutée au fichier et celui-ci chargé, il est possible de demander qui est ami avec qui, dans la console :

```
?- amis(X, Y).
X = chat,
Y = fromage ;
false.
```

Il n'y a donc potentiellement qu'un couple d'amis, le chat et le fromage (car le chat mange la souris qui mange le fromage).

Il est possible d'avoir plusieurs règles pour le même prédicat, tant qu'elles respectent toutes la même signature (donc le même nombre de paramètres) et qu'elles sont différentes.

9.2.5 Autres prédicats utiles

Il est aussi possible d'indiquer qu'une **règle échoue** si certaines conditions sont remplies avec le mot-clé "`fail`".

Dans tous les cas, lorsqu'une règle échoue, si le prédicat en possède d'autres, le moteur va les tester aussi grâce au backtracking. Il existe cependant un moyen d'empêcher ce backtracking, pour éviter de tester de nouvelles règles lorsqu'on sait que la règle ne réussira jamais : l'**opérateur "cut"** représenté par un point d'exclamation.

Il existe aussi des moyens d'utiliser une mini base de données inclue dans le langage qui permet de se rappeler de faits juste inscrits, de manière à ne pas avoir à les retrouver si d'autres règles en ont besoin. À l'inverse, il est possible de supprimer les faits enregistrés. Ces prédicats sont la série des `assert` (`assert`, `assertz`, `asserta`) et des `retract` (`retract` et `retractall`). Ils seront utilisés dans l'exemple. Les entrées/sorties avec l'utilisateur utilisent les prédicats de Prolog `read` et `write`.

Enfin Prolog contient plusieurs prédicats pour manipuler des listes, les créer, les parcourir...

9.3 Codage du problème des formes géométriques

Un nouveau projet est créé pour les noms des polygones. Comme pour la version en C#, cet exemple sera limité aux triangles et quadrilatères, mais il est facile de compléter les règles.

Plusieurs prédicats correspondent aux informations sur la forme : `cotesEgaux X)` indiquant le nombre de côtés égaux en taille, `angleDroit(X)` valant oui ou non et indiquant la présence d'au moins un angle droit, `cotesParallèles(X)` indiquant le nombre de côtés parallèles entre eux (0, 2 ou 4) et `ordre(X)` indiquant le nombre de côtés.

Le nom de la forme sera sous la forme `nom(X)`.

Il est donc possible d'écrire les différentes règles en suivant la syntaxe de Prolog :

```
%********** Rules **********
% Triangles
nom(triangle) :-
    ordre(3).

nom(triangleIsocele) :-
    nom(triangle),
    cotesEgaux(2).

nom(triangleRectangle) :-
    nom(triangle),
    angleDroit(oui).

nom(triangleRectangleIsocele) :-
    nom(triangleIsocele),
    nom(triangleRectangle).

nom(triangleEquilateral) :-
    nom(triangle),
    cotesEgaux(3).

% Quadrilateres
nom(quadrilatere) :-
    ordre(4).

nom(trapeze) :-
```

```
    nom(quadrilatere),
    cotesParalleles(2).

nom(parallelogramme) :-
    nom(quadrilatere),
    cotesParalleles(4).

nom(rectangle) :-
    nom(parallelogramme),
    angleDroit(oui).

nom(losange) :-
    nom(parallelogramme),
    cotesEgaux(4).

nom(carre) :-
    nom(losange),
    nom(rectangle).
```

On retrouve les mêmes règles que pour le code C#, cependant la lecture de celles-ci pour un novice est un peu plus complexe à cause de la syntaxe.

La gestion des faits sera aussi plus compliquée. Deux cas se poseront :

- Le fait est présent en mémoire : soit il a la bonne valeur et il est validé, soit il a la mauvaise valeur et on s'arrête ici en disant que la règle a échoué (et on ne cherche pas d'autres moyens de la résoudre).
- Le fait n'est pas présent en mémoire : on le demande à l'utilisateur, lit sa réponse, l'enregistre en mémoire, et regarde si la valeur répondue est celle attendue.

Un prédicat `memory` est créé. Celui-ci prend deux paramètres : le nom de l'attribut et sa valeur. Comme à l'origine il n'y a rien en mémoire, et que nous allons ajouter des faits au fur et à mesure, ce prédicat doit être indiqué comme étant dynamique dans le fichier de règles :

```
:- dynamic memory/2.
```

Pour traiter les valeurs des faits, un prédicat `ask` est créé. Celui-ci prend trois paramètres :

- Le fait que l'on cherche.
- La question à poser à l'utilisateur.
- La réponse obtenue.

Il a trois règles associées (à bien garder dans cet ordre-là). La première correspond au cas où le fait est déjà présent en mémoire, et avec la bonne valeur. La question à poser n'est alors pas importante vu que la réponse est déjà connue, et sera une variable anonyme de la règle. Dans ce cas, la règle fonctionne :

```
ask(Pred, _, X) :-
    memory(Pred, X).
```

Pour la deuxième règle, on sait qu'on l'évaluera uniquement si la première échoue. On cherche donc une valeur (anonyme) en mémoire. Si on en trouve une, cela signifie que le fait est déjà défini mais avec une valeur différente de celle attendue. La règle va donc échouer, et on ajoute le `cut` pour être sûr de ne pas évaluer une autre règle.

```
ask(Pred, _, _) :-
    memory(Pred, _),
    !,
    fail.
```

Enfin, la troisième règle concerne le cas où le fait n'a pas été trouvé en mémoire jusque-là (par les deux premières règles). La question est posée avec `write`, la réponse lue avec `read`, et le fait enregistré en mémoire avec `asserta` (qui ajoute en début). Enfin, on regarde si la valeur obtenue est la valeur attendue. Si oui, la règle réussit, sinon elle échoue, et comme il n'y a pas d'autres règles, le prédicat demandé est considéré comme faux.

```
ask(Pred, Question, X) :-
    write(Question),
    read(Y),
    asserta(memory(Pred, Y)),
    X == Y.
```

Les quatre faits qui peuvent être demandés à l'utilisateur en se basant sur `ask` sont ajoutés aux règles :

```
cotesEgaux(X) :- ask(cotesEgaux, 'Combien la figure a-t-elle de
côtés égaux ? ', X).

angleDroit(X) :- ask(angleDroit, 'La figure possède-t-elle des
angles droits (oui, non) ? ', X).

cotesParalleles(X) :- ask(cotesParalleles, 'Combien la figure a-
t-elle de côtés parallèles (0, 2 ou 4) ? ', X).

ordre(X) :- ask(ordre, 'Combien de côtés ? ', X).
```

Enfin une dernière règle permettant de résoudre un cas est ajoutée. Celle-ci va d'abord effacer tous les faits enregistrés en mémoire, puis demander grâce au prédicat Prolog "`findAll`" toutes les valeurs X qui peuvent être unifiées avec `nom(X)` (ce qui revient à demander tous les noms de la forme). Le résultat est mis dans la variable R qui est enfin affichée.

En Prolog, cette règle s'écrit donc :

```
solve :-
    retractall(memory(_,_)),
    findall(X, nom(X), R),
    write(R).
```

Le fichier de règles est terminé, ainsi que le programme. Il ne contient qu'une cinquantaine de lignes de code, contre plusieurs centaines pour le programme en C#. Pour l'utiliser, il suffit dans la console d'appeler le prédicat `solve`.

Voici des exemples de dialogues obtenus (pour un triangle rectangle isocèle puis un rectangle) :

```
?- solve.
Combien de côtés ? 3.
Combien la figure a-t-elle de côtés égaux ? 2.
La figure possède-t-elle des angles droits (oui, non) ? Oui.
[triangle,triangleIsocele,triangleRectangle,triangleRectangleIsocele]
true.

?- solve.
Combien de côtés ? 4.
Combien la figure a-t-elle de côtés parallèles (0, 2 ou 4) ? 4.
La figure possède-t-elle des angles droits (oui, non) ? Oui.
```

```
Combien la figure a-t-elle de côtés égaux ? 2.
[quadrilatere,parallelogramme,rectangle]
true.
```

Remarque

Attention : pour valider une réponse demandée par `read`*, il faut terminer la ligne par un point.*

9.4 Codage du problème des huit reines

9.4.1 Intérêt du chaînage arrière

Prolog utilise un chaînage arrière, c'est-à-dire qu'il part d'un but à atteindre et cherche toutes les règles lui permettant d'y arriver. Dans l'exemple précédent, on devait donc lui indiquer que l'on cherchait à associer un nom à la forme. Un chaînage avant aurait pu être plus simple à mettre en œuvre (mais il n'est pas intégré à Prolog).

Le chaînage arrière est surtout utile lorsqu'il y a de nombreux bactracking à faire, pour tester d'autres solutions lorsqu'une échoue. Le problème des 8 reines en est un cas typique. En effet, on va chercher à positionner nos reines, et on tentera d'autres possibilités jusqu'à en obtenir une qui fonctionne. Il faut alors se rappeler tous les choix déjà tentés pour ne plus les recommencer.

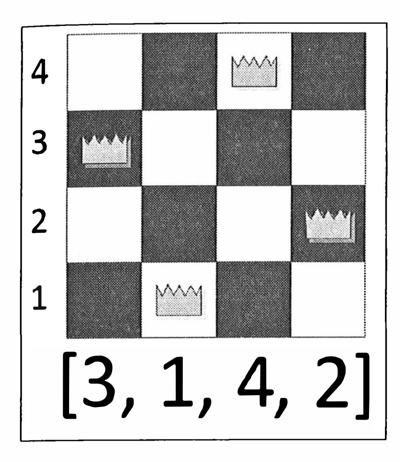
Écrire ce programme en C# est faisable. Il n'est cependant pas très lisible et s'éloignera grandement de la conception d'un système expert, les règles disparaissant au profit de boucles permettant de tester les différentes positions.

9.4.2 Étude du problème

De plus, le code sera une version générique permettant de résoudre le problème des N reines (il s'agit donc de placer N reines sur un échiquier de NxN cases). Cette solution n'est cependant pas complètement optimisée pour conserver la lisibilité, et les temps de calculs à partir de N=15 deviennent souvent trop importants pour des machines classiques (les codes Prolog les plus optimisés permettent de dépasser N=20, mais aucun n'arrive encore à répondre dans un temps acceptable pour N>30).

Avant de passer au codage, il est important de comprendre la logique de ce problème. Nous avons déjà vu dans ce chapitre qu'il était important de bien choisir la représentation de la solution. Dans ce cas, il s'agira donc d'une liste de N valeurs, toutes différentes. Cela permet de s'assurer que chaque reine est sur une colonne différente (donnée par la position dans la liste) et une ligne différente (car tous les nombres seront différents).

Ainsi, sur un échiquier de 4x4, on pourra obtenir la solution [3, 1, 4, 2] correspondant à la disposition suivante :



9.4.3 Règles à appliquer

On doit donc s'assurer que les nombres de la liste solution sont des permutations de la liste [0,..., N]. Ici, [3, 1, 4, 2] est bien une permutation de [1, 2, 3, 4]. Prolog nous permet facilement de construire la liste des entiers de 1 à N grâce au prédicat `numlist`, et d'en faire des permutations via le prédicat `permutation`.

La représentation ayant éliminé les problèmes des lignes et des colonnes, il ne restera donc à vérifier que le fait que deux reines ne sont pas en conflit sur une même diagonale. Pour chaque reine, il va donc falloir vérifier qu'elle n'est pas sur la même diagonale que toutes les reines suivantes.

Il y a donc une double boucle : il faut tester toutes les reines de la solution, et les comparer à toutes les reines restantes. Deux prédicats différents seront nécessaires.

Le parcours des listes est uniquement récursif en Prolog : il va donc falloir déterminer la condition d'arrêt, puis le cas général qui doit rappeler le prédicat en cours. De plus, Prolog permet de ne parcourir une liste qu'en séparant l'élément de tête de la suite de la liste. Ainsi, [T | Q] représente une liste commençant par l'élément T, suivi de la liste Q.

Les conditions d'arrêt se feront donc quand la liste des éléments suivants est vide : une reine ne peut pas entrer en conflit avec une liste vide d'autres reines.

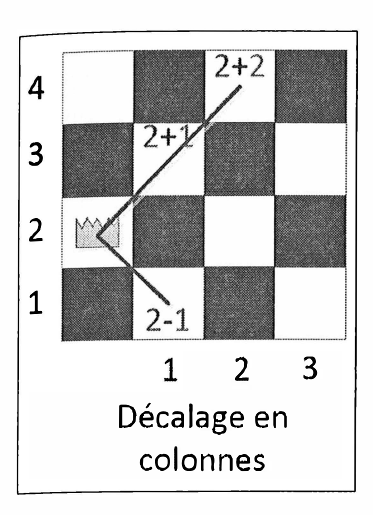
9.4.4 Règles de conflits entre reines

L'étude du problème ayant été effectuée, il est possible de passer à l'écriture du problème en Prolog.

Le premier prédicat doit indiquer si la reine en cours est en conflit avec la liste de reines fournie, et qui commence à Col colonnes plus loin. Ainsi, dans l'exemple donné pour les 4 reines [3, 1, 4, 2], on vérifie si la reine positionnée en 3 est en conflit avec la reine positionnée en 1 à 1 colonne d'écart, puis avec la reine située en 4 à 2 colonnes d'écart, puis enfin la reine située en 2 à 3 colonnes d'écart.

Ce nombre de colonnes est important. En effet, deux reines sont sur une même diagonale si le nombre de la reine en cours plus la différence de colonnes correspond à la position de la reine testée pour une diagonale montante (il faut enlever le nombre de colonnes pour la diagonale descendante).

Le schéma suivant indique un exemple de diagonale pour une reine positionnée en deuxième ligne : on voit qu'il faut ajouter à la ligne en cours la différence de colonnes pour trouver la case sur la même diagonale montante (ou la soustraire pour la diagonale descendante).



Ainsi, dans [3, 1, 4, 2], aucune reine n'est en conflit avec la reine en 3. En effet, 3+1 et 3-1 sont différents de 1 (la position de la reine à 1 colonne), 3+2 et 3-2 sont différents de 4 (la position de la reine à 2 colonnes) et 3+3 et 3-3 sont différents de 2 (la position de la reine à 3 colonnes).

Le cas d'arrêt est simplement la liste vide, indiquant qu'il n'y a aucun conflit et que la position fonctionne.

On a donc les deux règles suivantes pour le prédicat `diagReine` qui prend en premier paramètre la reine en cours, suivie de la liste des autres reines, et enfin l'écart en colonnes (le symbole =\= indique la différence) :

```
diagReine(_, [], _) :-
     true.

diagReine(Reine, [T|Q], Col) :-
     (Reine + Col) =\= T,
     (Reine - Col) =\= T,
     diagReine(Reine, Q, Col+1).
```

On sait maintenant si une reine précise entre en conflit avec les autres. Il faut donc boucler pour parcourir toutes les reines. On procède par récursivité : une reine n'est pas en conflit s'il ne reste plus de reines derrière, et si une reine donnée n'est pas en conflit alors on teste avec la liste diminuée de cette reine. Voici donc le prédicat `diagsOK` qui prend en paramètre la liste actuelle :

```
diagsOK( [ _ | [] ] ) :-
    true.

diagsOK([Tete | Queue]) :-
     diagReine(Tete, Queue, 1),
     diagsOK(Queue).
```

9.4.5 But du programme

Les diagonales sont maintenant testées, il n'y a plus qu'à écrire le but nommé "`reines`" et qui prend en paramètre le nombre de reines à placer. Celui-ci doit donc créer une liste de 1 à N, qui nous servira de base, puis chercher pour toutes les permutations de cette liste celles qui respectent les contraintes des diagonales :

```
reines(N, Res) :-
    numlist(1,N,Base),
    permutation(Res,Base),
    diagsOK(Res).
```

Ce coup-ci, on ne demande à afficher que la première solution trouvée. D'autres pourront être obtenues en appuyant sur ';' à chaque fois que Prolog propose une solution. En effet, s'il n'y a que deux solutions possibles pour le problème des 4 reines, il y en a 92 pour les 8 reines et le nombre augmente de manière exponentielle (bien qu'il n'existe pas de formule exacte pour le calculer).

9.4.6 Exemples d'utilisation

Voici donc quelques appels possibles en console, en demandant toutes les solutions pour N = 4 puis N = 6 et en demandant uniquement la première solution pour N = 8 :

```
?- reines(4, Res).
Res = [3, 1, 4, 2] ;
Res = [2, 4, 1, 3] ;
false.
```

```
?- reines(6, Res).
Res = [4, 1, 5, 2, 6, 3] ;
Res = [5, 3, 1, 6, 4, 2] ;
Res = [2, 4, 6, 1, 3, 5] ;
Res = [3, 6, 2, 5, 1, 4] ;
false.

?- reines(8, Res).
Res = [1, 7, 5, 8, 2, 4, 6, 3] .
```

Ce programme, non optimisé mais utilisant le backtracking natif de Prolog ne prend que 15 lignes, tout en définissant les règles permettant de dire que deux reines ne sont pas en conflit. L'intérêt des langages à programmation logique dans des cas comme celui-ci est donc assez évident.

Remarque

Les versions les plus rapides sont cependant codées en C (en quelques centaines de lignes). Prolog permet de simplifier l'écriture du programme, mais n'est pas toujours le plus efficace en termes de temps de réponse. Il existe cependant des solutions en Prolog moins lisibles mais beaucoup plus rapides que celle proposée ici.

10. Ajout d'incertitudes et de probabilités

Les systèmes experts vus jusqu'ici se basaient sur des règles sûres, et les faits étaient forcément vrais ou faux. Cependant, dans la réalité, les choses sont souvent plus complexes. Il faut donc penser à gérer les incertitudes.

10.1 Apport des incertitudes

Dans un système expert destiné à identifier des animaux en fonction de caractéristiques physiques, il peut être difficile d'estimer exactement le nombre de doigts aux pattes de l'animal ou la couleur de son ventre. Surtout s'il s'agit d'un prédateur ou d'un animal venimeux, il peut sembler difficile de l'examiner sous tous les angles pour répondre aux questions du système.

Dans ces cas-là, il peut être intéressant d'ajouter de l'incertitude sur les faits : l'utilisateur pourra donc dire qu'il lui semble que l'animal avait le ventre blanc, mais qu'il n'en est pas totalement sûr.

De plus, dans un système expert médical, il paraît dangereux de dire que si les symptômes d'une maladie sont des douleurs dans tout le corps, de la fièvre et une grande fatigue, alors il s'agit forcément d'une grippe. En effet, des maladies plus rares mais plus dangereuses pourraient se cacher derrière ces symptômes.

Ce coup-ci, ce sont les règles elles-mêmes qui sont incertaines : il y a de fortes chances que ce soit la grippe, mais ce n'est pas la seule explication possible.

Ces deux types d'incertitudes (sur les faits et les règles) peuvent être gérés par un système expert pour le rendre plus efficace.

10.2 Faits incertains

Pour les faits, une probabilité peut leur être ajoutée. Elle indique à quel point l'utilisateur est sûr de lui.

Ainsi, un fait sûr à 80 % indique que l'utilisateur a un petit doute sur le fait. À l'inverse, un fait sûr à 100 % indique qu'il est absolument certain.

Il est assez facile d'ajouter ces probabilités. Cependant, lors de l'application de règles, il faudra changer le fonctionnement du moteur d'inférences. Celui-ci commencera par calculer la probabilité des prémisses des règles. Il s'agira de la valeur minimum des différents faits.

Ainsi, si on a une règle du type "Si A et B alors C", et que A est vrai à 75 % et B à 85 %, on considérera que l'ensemble des prémisses est vrai à 75 %.

Une règle dont la probabilité est inférieure à 50 % ne sera généralement pas appliquée.

Le fait inféré prendra aussi pour valeur de certitude celle de la règle. Dans notre cas précédent, on rajouterait le fait C avec une valeur de 75 % à la base de faits.

De cette façon, on aura une propagation des différentes probabilités.

10.3 Règles incertaines

Comme pour les faits, des probabilités peuvent être ajoutées aux règles. On pourra ainsi dire qu'un diagnostic est vrai à 75 %, c'est-à-dire 3 fois sur 4.

Un fait inféré à partir de cette règle serait donc lui aussi vrai à 75 %, les faits inférés prenant comme probabilité celle de la règle qui l'a créé.

On peut bien évidemment cumuler les probabilités sur les faits et les règles. La probabilité d'un fait inféré est la probabilité des prémisses multipliée par la probabilité de la règle. Ainsi des prémisses vraies à 80 % dans une règle vraie à 75 % donneront un fait inféré qui sera vrai à 75*80/100 = 60 %.

Si un même fait inféré est obtenu de différentes façons (par exemple en appliquant plusieurs règles), il faut combiner les probabilités obtenues. Le calcul est ici plus complexe car il nécessite de prendre en compte les probabilités déjà obtenues.

En effet, si une première règle nous dit que le fait est sûr à 80 % et une autre que le fait l'est à 50 %, on ne peut pas seulement conclure qu'il est vrai à 80 % en prenant la valeur maximale. On dira alors que sur les 20 % non sûrs à l'issue de la première règle, la deuxième en comble 50 %, soit 10 % du total. Le fait inféré aura alors une probabilité de 90 % (80+10).

La formule permettant de calculer cette probabilité totale en fonction d'un fait vrai à une probabilité P_a et une nouvelle règle le produisant avec une probabilité P_b est :

$$P_{totale} = P_a + (1 - P_a) * P_b$$

On peut remarquer que l'ordre d'application des règles n'est pas important, le résultat obtenu étant toujours le même.

Si le besoin s'en fait sentir, il est donc possible d'intégrer ces probabilités à tous les niveaux, pour améliorer les systèmes experts produits.

11. Synthèse

Un système expert permet de partir de faits et d'y appliquer des règles pour obtenir de nouveaux faits, dits inférés. Ils remplacent ou complètent le savoir des experts selon les cas.

Ceux-ci sont composés d'une base de règles indiquant toutes les règles connues de l'expert, et qui permet d'arriver à des déductions. Ils se composent aussi d'une base de faits reprenant tout ce qui est connu de l'utilisateur ainsi que les faits inférés jusqu'ici. Une interface utilisateur permet de communiquer de manière claire avec les différentes personnes utilisant le système.

Le cœur du système expert est son moteur d'inférences. C'est lui qui va choisir et appliquer les règles, et lancer les interactions avec l'utilisateur. Il peut être à chaînage avant s'il part des faits pour en obtenir de nouveaux ou à chaînage arrière s'il part d'un but et cherche comment y arriver.

La création de ce moteur n'est pas toujours chose aisée, mais il est possible d'en implémenter dans tous les langages. Les moteurs à chaînage avant sont cependant bien souvent plus simples à coder, par exemple en C#. Des langages particuliers, issus de la programmation fonctionnelle ou logique, comme Prolog, permettent de simplifier la mise en place de systèmes experts. En effet, le moteur fait partie intégrante du langage.

Enfin, il est possible d'ajouter la gestion des incertitudes, autant au niveau des faits entrés par l'utilisateur que des règles, pour les domaines plus difficiles à modéliser ou dans des cas où l'utilisateur ne peut répondre avec certitude aux questions posées par le système.

Les systèmes experts, de par leur facilité de mise en œuvre, leur puissance, et leur facilité d'utilisation, se retrouvent aujourd'hui dans de nombreux domaines, que ce soit dans le diagnostic, l'estimation des risques, la planification et la logistique ou le transfert de connaissances et compétences.

Chapitre 2
Logique floue

1. Présentation du chapitre

La logique floue est une technique d'intelligence artificielle déterministe permettant de prendre des décisions. Elle permet ainsi d'avoir un comportement cohérent et reproductible en fonction de règles qui lui sont fournies. L'intelligence de cette technique se trouve dans sa capacité à gérer l'imprécision et à avoir un comportement plus souple qu'un système informatique traditionnel.

Ce chapitre commence par définir la notion d'imprécision, à ne pas confondre avec l'incertitude. Ensuite, les différents concepts sont abordés : les ensembles flous, les fonctions d'appartenance et les différents opérateurs en logique floue.

La partie suivante traite des règles floues et des étapes pour appliquer ces règles à un cas concret et en sortir un résultat utilisable (elles s'appellent respectivement fuzzification et défuzzification).

Le chapitre continue ensuite avec la présentation de différents domaines d'application de la logique floue, que l'on retrouve aujourd'hui de nos lave-linge à nos voitures en passant par les usines.

Enfin, la dernière partie consiste à montrer comment on peut implémenter un moteur de logique floue générique et évolutif en C#. Les différentes classes sont détaillées et le code complet est téléchargeable. Un exemple d'utilisation de ce moteur est aussi fourni. Une synthèse clôt ce chapitre : les principaux résultats y sont rappelés.

2. Incertitude et imprécision

Il est important de différencier deux concepts : l'incertitude et l'imprécision. En effet, chacun va être associé à une technique d'intelligence artificielle différente.

2.1 Incertitude et probabilités

L'**incertitude**, c'est évidemment, le contraire de la certitude. Par exemple, la règle "S'il va pleuvoir, alors je prendrai mon parapluie" est sûre (à 100 %) : se mouiller n'est pas agréable. Au contraire, l'énoncé "Demain, il devrait pleuvoir" est incertain : la météo a peut-être annoncé de la pluie, mais rien ne dit qu'il pleuvra vraiment. On peut dire que la probabilité qu'il pleuve est de 80 % par exemple. Tout énoncé dont la probabilité est différente de 100 % est incertain.

2.2 Imprécision et subjectivité

Au contraire, l'**imprécision** se manifeste lorsque l'on manque... de précision ! Dans les faits, cela se traduit par des énoncés qu'il est difficile d'évaluer : ils semblent subjectifs. Par exemple, dans la phrase "S'il fait très chaud, alors je ne mettrai pas de pull", l'imprécision se situe sur la notion de "très chaud".

On est certain que s'il fait "très chaud", je ne prendrai pas de pull, il n'y a donc pas d'incertitude. Mais fait-il "très chaud" à 35° ? Oui, sûrement. Et à 30° ? À la limite, on pourra statuer qu'il fait "très chaud" à partir de 30°. Mais dans ce cas-là, qu'en est-il à 29,5° ?

On se rend compte que pour un être humain, "très chaud" est une notion très floue : elle dépend des personnes, du lieu, du contexte... ainsi, très chaud pour un marseillais n'a certainement pas la même signification que pour un inuit. De plus, on ne sort pas un thermomètre pour l'évaluer, on se base sur le ressenti de notre peau. On est donc très imprécis. La majorité de nos décisions subit cette imprécision. C'est ainsi par exemple que l'on décide (ou non) de traverser une route hors des clous (en estimant la vitesse des voitures et leur distance) ou notre façon de nous habiller (en fonction du temps qu'il fait).

2.3 Nécessité de traiter l'imprécision

On peut imaginer un store géré par un système informatique classique. Si on lui donne la règle "si température supérieure ou égale à 25°, alors baisser le store, sinon le monter", on risque de se retrouver avec un store qui ne fera que monter et descendre s'il fait 25° avec un ciel légèrement nuageux.

En effet, à chaque nuage, la température pourrait descendre à 24.9° et le store se lèverait. Une fois le nuage passé, la température remonterait et le store descendrait. Le moteur serait donc en permanence activé. En réalité, ce qu'on aimerait lui dire c'est "S'il fait chaud dehors, alors baisser le store, sinon le monter", mais un ordinateur ne comprend pas le terme "chaud".

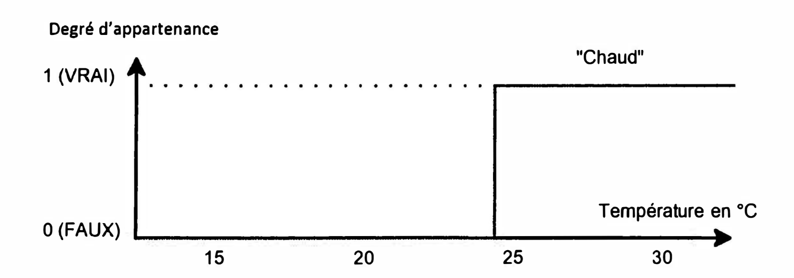
C'est justement pour gérer cette imprécision qu'est apparue la **logique floue** en 1965. Elle a été formalisée par Lotfi Zadeh comme une extension de la **logique booléenne** (la logique "classique" dans laquelle tout ne peut être que vrai ou faux). Elle nous permet de définir un énoncé, tout à fait certain, mais dont les données sur lesquelles il se base sont subjectives ou, en tout cas, imprécises.

3. Ensembles flous et degrés d'appartenance

Si on reprend l'exemple de notre store électrique et de la température, on aimerait définir le terme "chaud". On va donc définir pour quelles températures il fait chaud ou non.

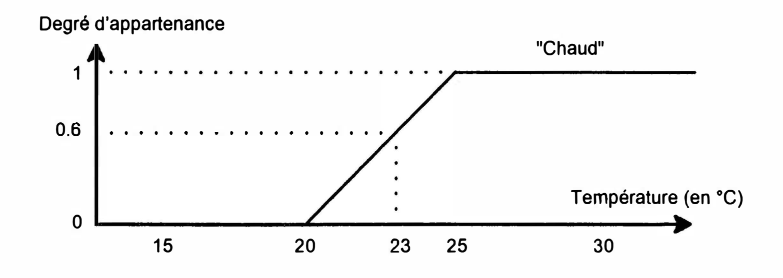
3.1 Logique booléenne et logique floue

En logique booléenne (la logique classique), une valeur peut seulement être vraie ou fausse. On doit définir une valeur précise qui sert de transition. Pour définir "chaud" pour notre store, cette valeur est 25°. Au-dessus de cette température, "chaud" est vrai, au-dessous, "chaud" est faux. Il n'y a pas de valeurs intermédiaires.



En logique floue, on va utiliser un **ensemble flou**. Il se différencie de l'ensemble booléen par la présence d'une "**phase de transition**", durant laquelle la variable se situe entre vrai et faux. Pour la température, entre 20° et 25°, il fait alors plus ou moins chaud.

Au-dessous de 20°C, il ne fait pas chaud, et au-dessus de 25°, il fait chaud à 100 %. Mais à 23°, il ne fait chaud qu'à 60 % (soit 0.6).

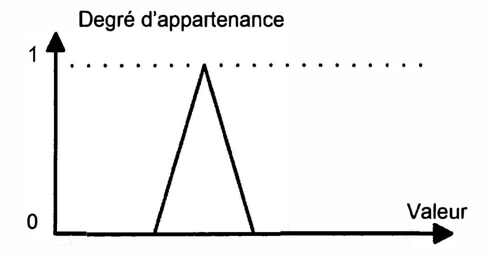


3.2 Fonctions d'appartenance

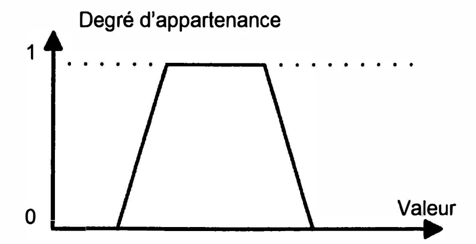
On voit ainsi qu'en logique booléenne, on ne travaille que sur les termes "vrai" ou "faux". On passe de faux (0 %) à vrai (100 %) à 25°. En logique floue, on rajoute des états intermédiaires : de 20° à 25° on va passer progressivement de faux à vrai. Ainsi, on peut lire sur le graphique que le **degré d'appartenance** (ou valeur de vérité) à "chaud" pour une température de 23° est de 0.6 soit 60 %. De même, à 24°, il fait "chaud" à 80 %.

La courbe indiquant les degrés d'appartenance est appelée **fonction d'appartenance**. Elle permet de définir un **ensemble flou**, possédant des limites qui ne sont pas nettes, mais progressives, comme un fondu. Ces ensembles flous peuvent utiliser différentes fonctions d'appartenance qui associent toutes un degré d'appartenance aux différentes valeurs possibles. Il existe cependant cinq fonctions plus classiques.

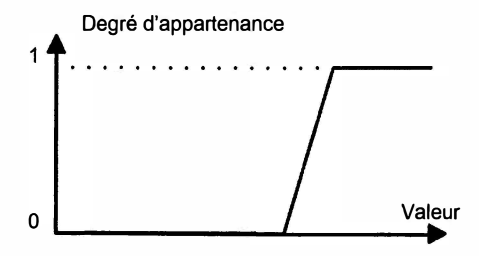
1. La fonction triangulaire : il n'y a qu'une seule valeur possédant un degré d'appartenance de 1, et on observe deux phases de transition linéaires (avant et après cette valeur).



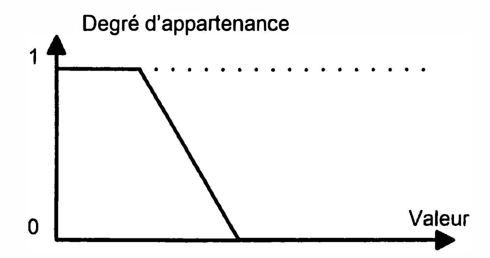
2. La fonction trapézoïdale : on observe un plateau pour lequel toutes les valeurs sont vraies, avec deux phases de transition linéaires avant et après le plateau.



3. Les fonctions 1/2 trapèze (à droite ou à gauche) : elles servent à représenter des seuils. Toutes les valeurs situées avant ou après une valeur donnée sont vraies à 100 %. Une phase de transition linéaire sépare ce plateau des valeurs entièrement fausses.

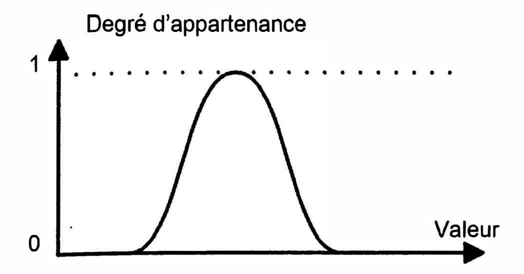


1/2 trapèze à droite

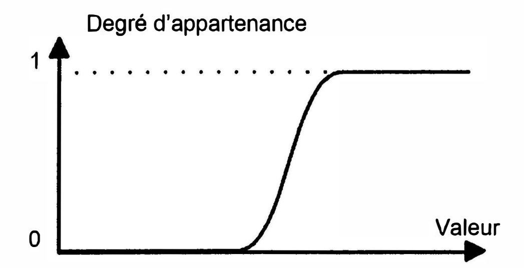


1/2 trapèze à gauche

4. La fonction gaussienne (plus connue sous le nom de "cloche") : elle reprend le principe de la fonction triangulaire, en éliminant les angles, ce qui permet des transitions encore plus douces.



5. La fonction sigmoïde : elle reprend la fonction 1/2 trapèze, en remplaçant là encore les angles et les transitions linéaires par une courbe plus douce.



Quelle que soit la fonction choisie, il reste une constante : chaque fonction d'appartenance se doit d'associer un degré compris entre 0 et 1 à chaque valeur potentielle du domaine. Mathématiquement, les fonctions sont donc continues sur tout l'ensemble de définition.

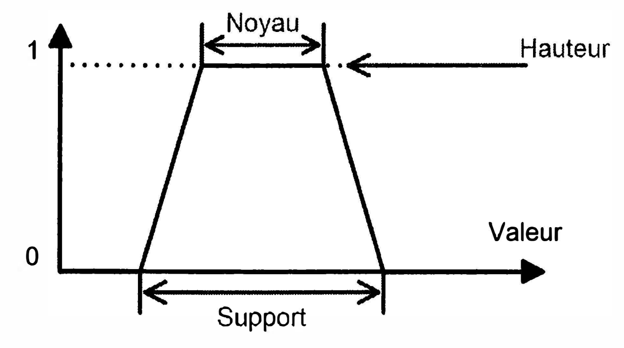
Remarque

La forme des fonctions d'appartenance est cependant libre. Il est donc tout à fait possible d'utiliser des fonctions différentes de celles-ci, mais c'est dans la pratique très rare. En effet, ces fonctions sont bien connues tout comme leurs propriétés mathématiques, ce qui simplifie les calculs.

3.3 Caractéristiques d'une fonction d'appartenance

Les différentes fonctions d'appartenance sont caractérisées par :

- La **hauteur** : c'est le degré maximal d'appartenance que l'on peut obtenir. Dans une grande majorité des cas, on ne considérera que des fonctions dont la hauteur vaut 1. Ces fonctions sont dites normalisées.
- Le **support** : c'est l'ensemble des valeurs pour lesquelles le degré d'appartenance est différent de 1. Cela représente donc toutes les valeurs pour lesquelles le terme est plus ou moins vrai (et donc n'est pas faux).
- Le **noyau** : c'est l'ensemble des valeurs pour lesquelles le degré d'appartenance vaut 1. Il est donc inclus dans le support, et correspond uniquement aux valeurs vraies à 100 %.



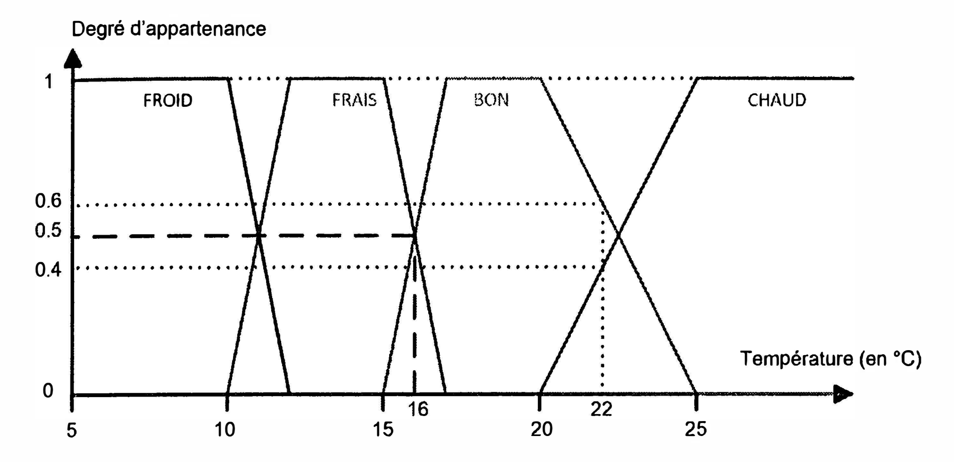
Pour le terme "chaud" défini précédemment, on a donc une hauteur de 1 (la fonction est normalisée), le noyau est l'ensemble [25 ; +∞] (toutes les températures supérieures ou égales à 25°C sont au moins totalement vraies), et le support est l'ensemble [20 ; +∞] (toutes les températures supérieures ou égales à 20° sont au moins partiellement vraies).

3.4 Valeurs et variables linguistiques

"Chaud" est appelé **valeur linguistique**. Il s'agit donc d'une valeur qui représente un terme du langage courant.

On peut imaginer définir plusieurs valeurs linguistiques, par exemple : "froid", "frais", "bon" et "chaud". L'ensemble de ces valeurs va alors définir la "température", qui sera appelée **variable linguistique**.

Pour notre variable linguistique "température", on peut donc obtenir le schéma global suivant, représentant les différentes valeurs linguistiques.



Sur la figure, on peut voir que quatre valeurs linguistiques ont été définies : "froid", "frais", "bon", "chaud".

Il est important de comprendre comment on peut lire les degrés d'appartenance d'une valeur numérique, par exemple 16°. Sur le schéma, on voit en tirets que pour 16°, on croise deux courbes : celle représentant la valeur "frais" et celle représentant la valeur "bon". À 16°, il fait frais et bon, mais il ne fait pas du tout froid ni chaud. La lecture du graphique indique que "frais" et "bon" sont chacun vrais à 50 %.

De la même façon, pour 22° (en pointillés), on voit qu'il ne fait pas du tout "froid" ni "frais", mais qu'il fait "bon" à 60 % (ou 0,6) et "chaud" à 40 %.

Remarque

Dans notre exemple, les valeurs linguistiques sont définies de telle sorte que la somme des degrés d'appartenance fasse toujours 1 pour une valeur donnée. Ce n'est pas une obligation, mais c'est une bonne pratique lorsque l'on définit une variable linguistique.

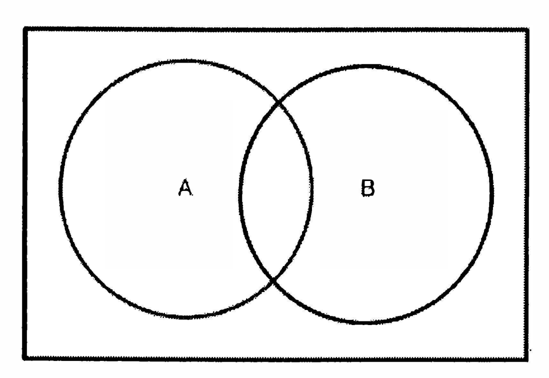
4. Opérateurs sur les ensembles flous

En logique classique, on a trois opérateurs de composition élémentaires : l'union (OU), l'intersection (ET) et la négation (NON). Ces opérateurs sont aussi nécessaires en logique floue, en particulier pour pouvoir composer des valeurs linguistiques (par exemple "frais OU bon") et écrire des règles.

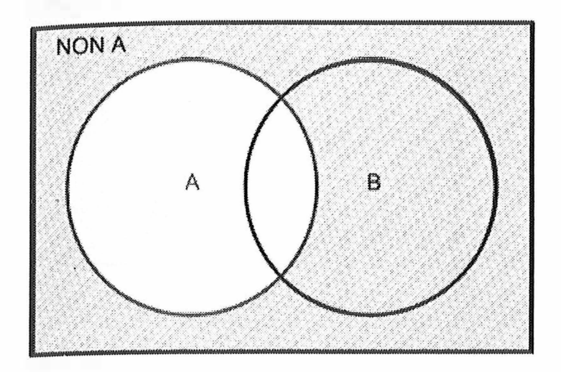
4.1 Opérateurs booléens

En logique booléenne, ces trois opérateurs peuvent se représenter grâce à des **diagrammes de Venn**. Dans ceux-ci, les ensembles sont représentés par des cercles.

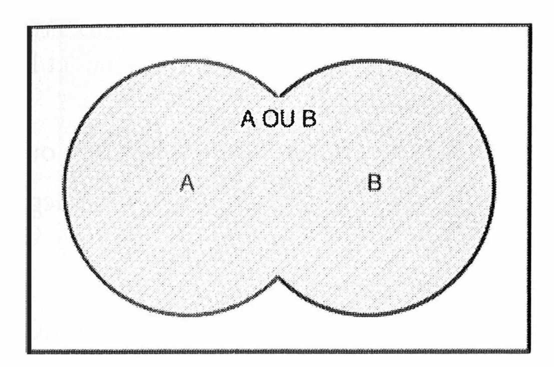
Ici, on a deux ensembles, A et B, qui se chevauchent en partie :



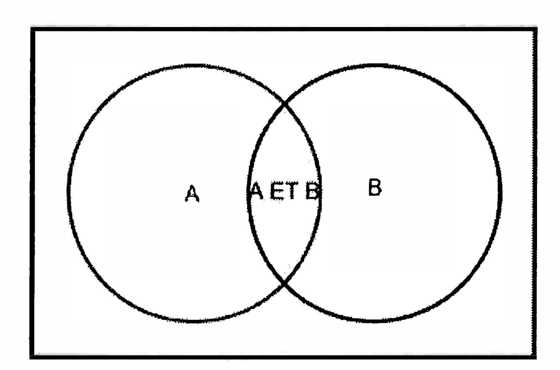
La négation de l'ensemble A est dite **NON A** et s'écrit $\overline{A}$. Elle représente l'ensemble des valeurs qui n'appartiennent pas à A. Elle est ici représentée par la zone finement hachurée :



L'union de A et B est dite **A OU B** et se note $A \cup B$. Elle représente l'ensemble des valeurs appartenant à l'un ou l'autre des ensembles. Elle est ici représentée par la zone finement hachurée :



Enfin, l'intersection lue **A ET B** et notée $A \cap B$, représente l'ensemble des valeurs appartenant aux deux ensembles en même temps. Elle est ici représentée par la zone finement hachurée commune aux ensembles :



4.2 Opérateurs flous

En logique floue, il n'y a pas de démarcation nette entre ce qui est faux et ce qui est juste. Les opérateurs classiques ne peuvent donc plus s'appliquer et les diagrammes de Venn ne sont plus adaptés.

Il faut alors travailler sur les fonctions d'appartenance des ensembles flous. Pour la suite, cette fonction est notée μ_A pour l'ensemble flou A. Le degré d'appartenance d'une valeur numérique particulière x s'écrit donc $\mu_A(x)$.

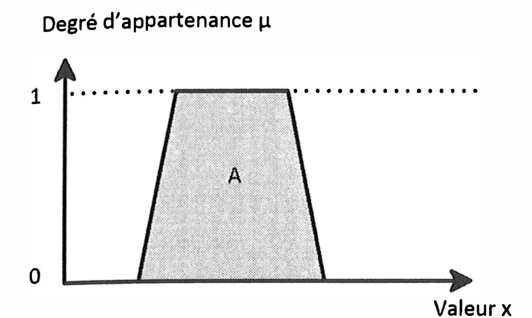
4.2.1 Négation

Pour la **négation floue**, NON A est défini comme l'ensemble flou ayant pour fonction d'appartenance la fonction définie par :

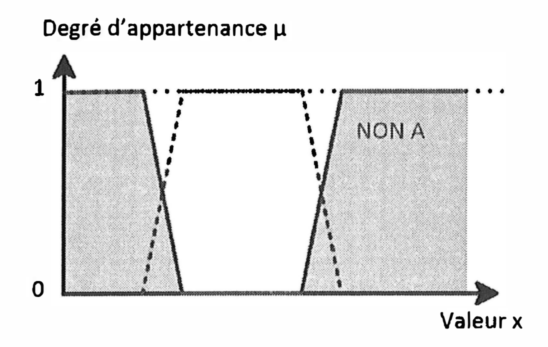
$$\mu_{\bar{A}}(\mathrm{x}) = 1 - \mu_A(x)$$

Cela signifie que si 22° est considéré comme "chaud" à 0,4 (soit 40 %), alors il est considéré comme "NON chaud" à 1-0,4=0,6 (soit 60 %).

Il est possible de représenter graphiquement cette négation. Tout d'abord, définissons un ensemble flou A dont la fonction d'appartenance est la suivante :



L'ensemble flou $\bar{A}$ a donc pour fonction d'appartenance la fonction suivante (celle de A est rappelée avec des tirets) :



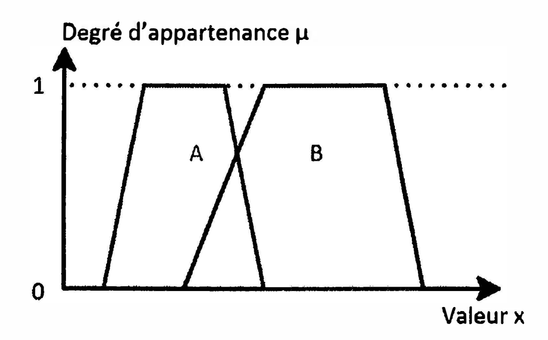
On remarque que les plateaux sont inversés : un plateau à 1 se retrouve à 0 et vice-versa. De plus, les transitions ont été échangées.

Remarque

Géométriquement, on peut s'apercevoir que les deux courbes sont les symétriques l'une de l'autre par rapport à un axe d'équation : degré = 0,5.

4.2.2 Union et intersection

Pour étudier ces deux opérateurs, deux ensembles flous A et B sont définis ainsi que leurs fonctions d'appartenance :

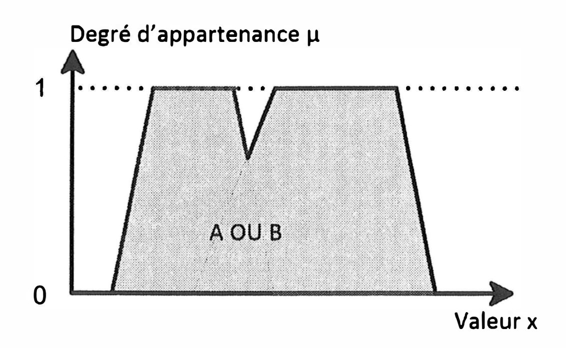


Il existe plusieurs possibilités pour calculer l'union ou l'intersection de ces deux ensembles. La plus courante (et la plus simple à mettre en œuvre) est d'utiliser les opérateurs définis par Zadeh.

L'union A ∪ B est définie par la fonction d'appartenance :

$$\mu_{A \cup B}(x) = \max(\mu_A(x), \mu_B(x))$$

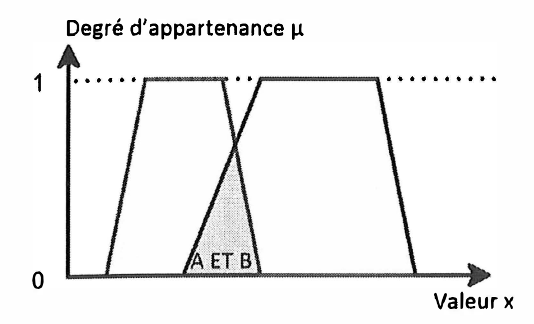
Cela revient à garder l'aire présente sous les deux courbes. En effet, pour chaque valeur, c'est la hauteur maximale entre les deux fonctions d'appartenance qui est conservée. On obtient alors le schéma suivant :



Pour l'intersection A ∩ B, la fonction d'appartenance est définie par :

$$\mu_{A \cap B}(x) = \min(\mu_A(x), \mu_B(x))$$

Ce coup-ci, seule l'aire commune aux deux ensembles est conservée, car pour chaque valeur c'est la hauteur minimale qui donne le degré d'appartenance. Cela donne alors :



Ces opérateurs sont les plus proches de ceux de la logique booléenne vu qu'ils reviennent à appliquer les opérateurs classiques non plus sur les diagrammes de Venn, mais sur les courbes des fonctions d'appartenance.

Remarque

D'autres auteurs ont proposé des opérateurs d'union et d'intersection différents. On peut citer les opérateurs de Łukasiewicz, qui sont la variante la plus utilisée. Leur formulation est cependant plus complexe et ne sera pas traitée ici. Retenez surtout que les opérateurs de Zadeh ne sont pas l'unique choix possible.

Grâce à ces opérateurs ensemblistes, nous allons pouvoir écrire des règles floues et, plus tard, les évaluer pour prendre des décisions.

5. Création de règles

5.1 Règles en logique booléenne

Dans un système classique, comme le contrôle du store du début de ce chapitre, une règle s'exprime sous la forme :

```
SI (condition précise) ALORS action
```

On a par exemple :

```
SI (température ≥ 25°) ALORS baisser le store
```

On peut aussi concevoir des règles plus complexes. Par exemple, on pourrait prendre en compte l'éclairage extérieur, qui se mesure en lux (car s'il y a du soleil, il faut s'en protéger). Il va de 0 (nuit noire sans étoile ni lune) à plus de 100 000 (éclairage direct du soleil). Un ciel nuageux de jour correspond à un éclairage entre 200 et 25 000 lux environ (selon l'épaisseur de nuages).

Dans notre application de contrôle de store, on pourrait donc créer la règle :

```
SI (température ≥ 25° ET éclairage ≥ 30 000 lux) ALORS baisser le store
```

Cela pose cependant des problèmes lorsque les températures mesurées sont proches des valeurs limites.

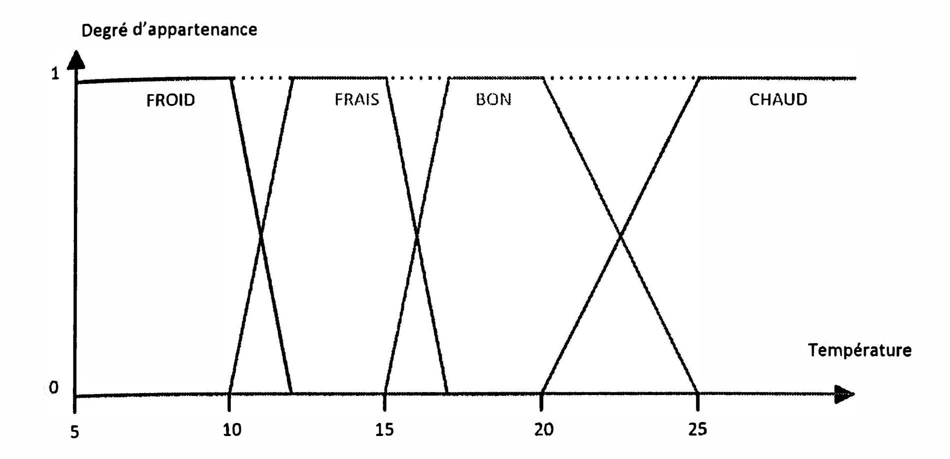
5.2 Règles floues

Dans un système flou, les règles utilisent des valeurs floues au lieu des valeurs numériques. On note les expressions utilisées dans les règles sous la forme :

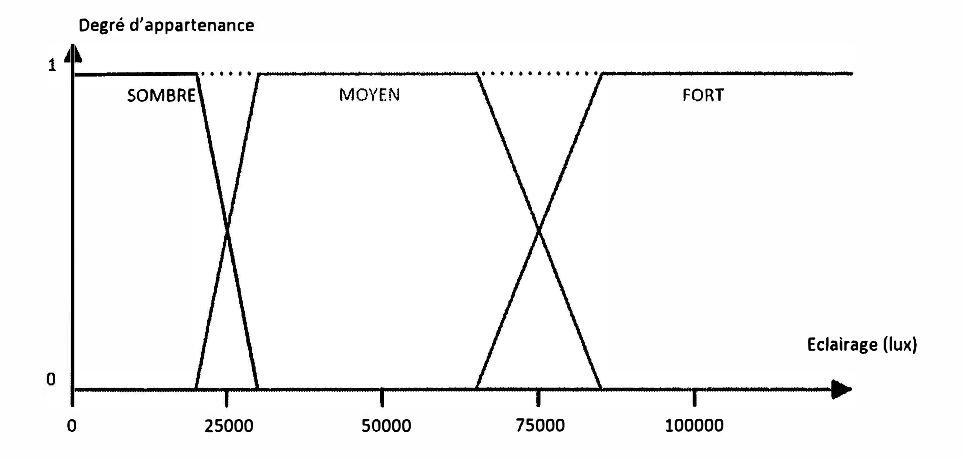
```
"Variable linguistique" EST "valeur linguistique"
```

Nous allons donc définir trois variables linguistiques : la température, l'éclairage et la hauteur du store.

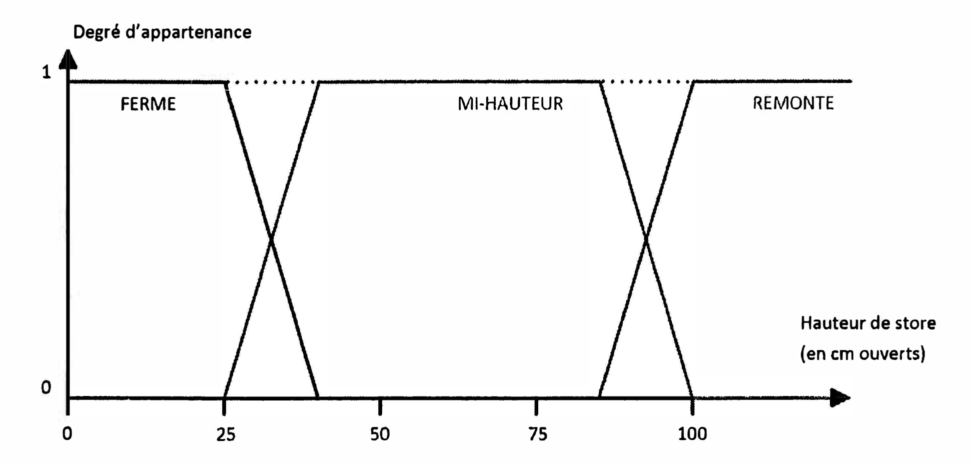
On commence par la température. Pour cela, on reprend simplement le schéma fait précédemment, les températures étant exprimées en °C.



L'éclairage représente la puissance du soleil sur la fenêtre. Celui-ci s'exprime en lux.



Enfin, la hauteur du store est mesurée en cm ouverts sur une fenêtre classique de 115 cm. À 0, le store est complètement fermé, et à 115 il est donc complètement remonté.



La température et l'éclairage sont des variables linguistiques en entrée : elles se mesurent directement et sont à la source de nos décisions. Au contraire, la hauteur du store est une variable de sortie : c'est la décision à prendre.

Voici un exemple de règle que l'on pourrait définir :

```
SI (température EST chaud ET éclairage EST fort) ALORS hauteur de
store EST fermé
```

Bien évidemment, un système flou peut posséder de nombreuses règles. Le plus simple pour les représenter lorsque deux variables linguistiques sont en entrée est un tableau à double entrée, qui indique pour chaque combinaison de valeurs la valeur de la variable de sortie (pour nous la hauteur de store).

Le tableau suivant indique donc les 12 règles, qui ont été numérotées pour plus de facilité de R1 à R12. Chacune d'elle correspond à un cas de température et d'éclairage, et indique la décision à prendre. Par exemple, s'il fait froid avec un fort éclairage, alors le store sera remonté pour essayer de profiter un maximum des rayons du soleil (règle R3).

Éclairage → Température ↓	Sombre	Moyen	Fort
Froid	R1. Remonté	R2. Remonté	R3. Remonté
Frais	R4. Remonté	R5. Remonté	R6. Mi-hauteur
Bon	R7. Remonté	R8. Mi-hauteur	R9. Fermé
Chaud	R10. Remonté	R11. Mi-hauteur	R12. Fermé

Nos règles vont donc essayer de maximiser le confort de l'utilisateur, en sachant que s'il fait trop chaud et que le soleil est fort, il est mieux d'être à l'ombre, alors que s'il fait froid, il faut profiter du soleil s'il est là.

6. Fuzzification et défuzzification

6.1 Valeur de vérité

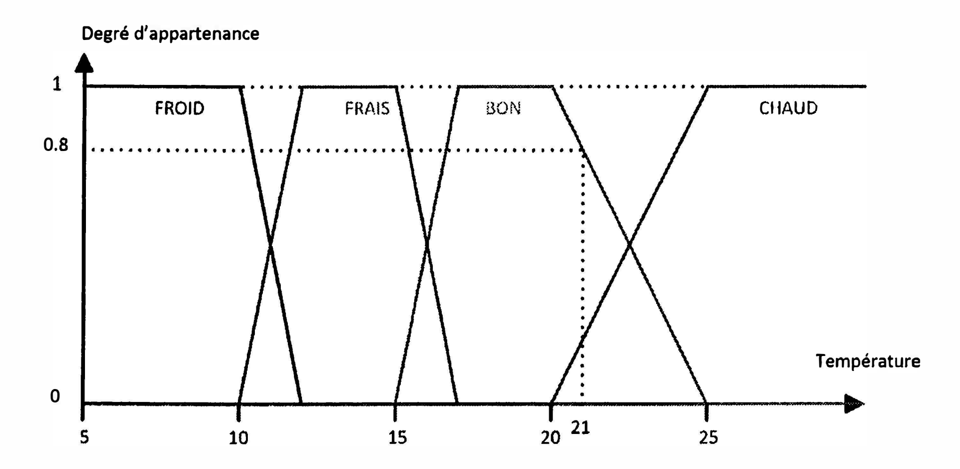
Les différentes règles possèdent toutes une implication (la clause ALORS). Il va donc falloir exprimer à quel point la règle doit être appliquée en fonction des valeurs numériques mesurées : c'est l'étape de **fuzzification**.

Nous allons nous intéresser à la règle R8 :

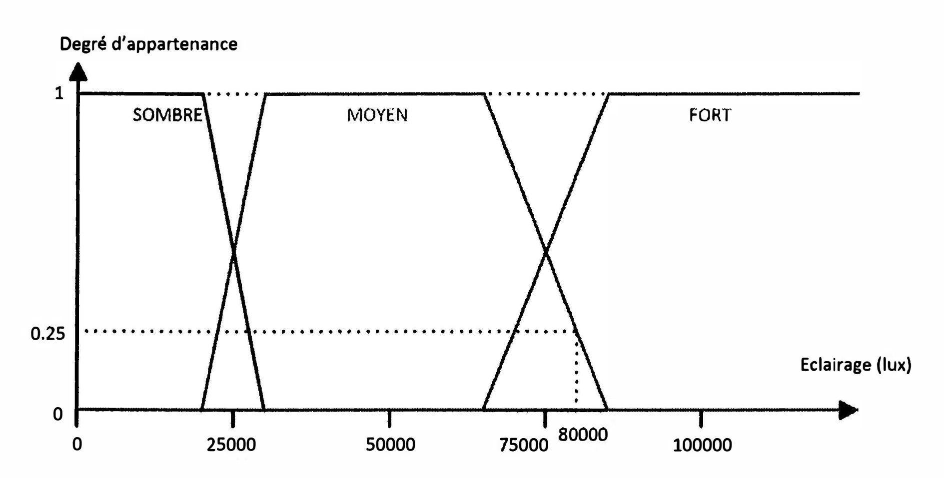
```
SI température EST bon ET éclairage EST moyen ALORS store EST
à mi-hauteur
```

Nous souhaitons savoir à quel point cette règle s'applique pour une température de 21°C et un éclairage de 80 000 lux.

On va donc commencer par chercher à quel point il fait BON. La figure suivante nous indique qu'à 21°C, il fait bon à 80 %.



Nous cherchons ensuite à quel point l'éclairage est MOYEN pour 80000 lux. On peut lire qu'il l'est à 25 % :



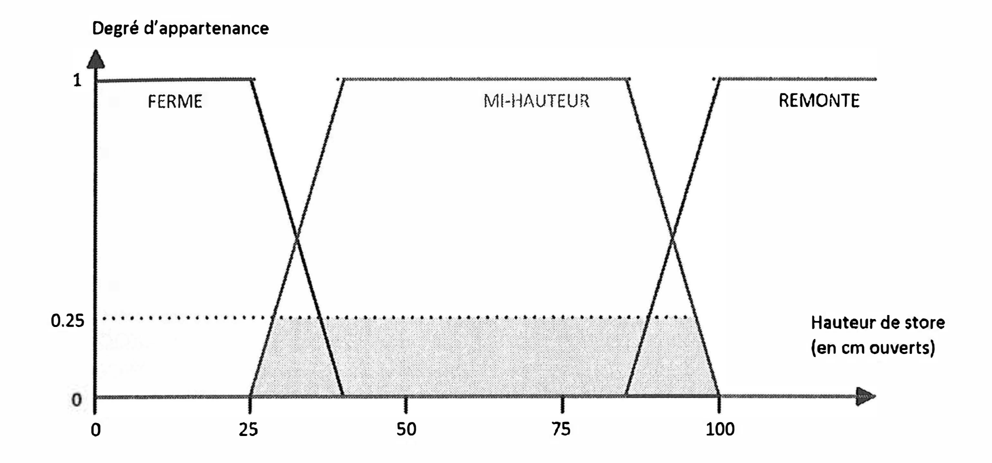
La règle contient donc une partie vraie à 80 % et une partie vraie à 25 %. On dira que la règle entière est vraie à 25 %, la valeur minimale (opérateur ET). C'est donc le terme le moins vrai qui donne la valeur de vérité d'une règle entière.

6.2 Fuzzification et application des règles

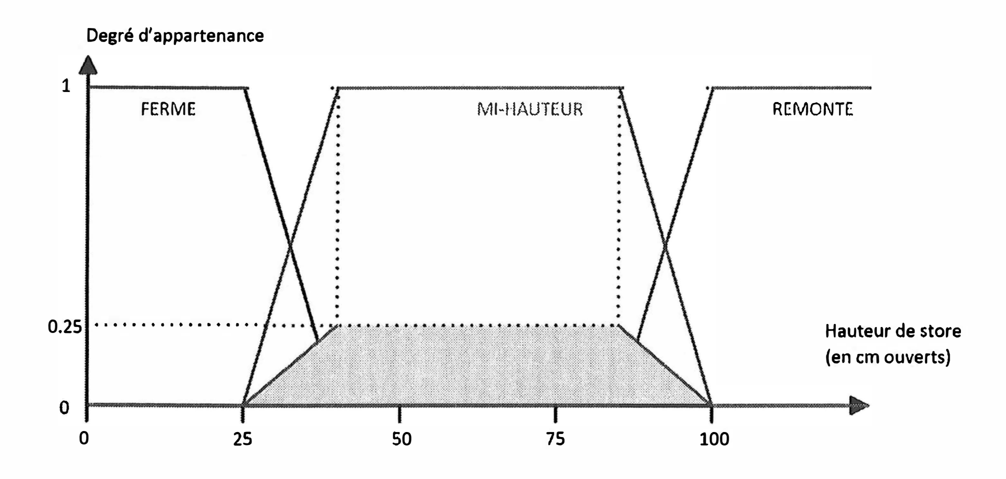
On cherche maintenant à savoir quel sera le résultat de cette règle. Elle nous dit que le store doit être à mi-hauteur. On sait que la règle s'applique à 25 %.

On a alors de nombreux choix pour l'opérateur d'implication afin de déterminer l'ensemble flou résultant. Nous allons nous intéresser à deux d'entre eux : l'implication d'après Mamdani et celle d'après Larsen. Ce qui change entre les deux, c'est la forme de l'ensemble flou d'arrivée.

Pour l'opérateur d'implication de Mamdani, l'ensemble flou est troqué à la valeur de vérité de la règle. Pour notre règle R8 qui s'applique à 25 %, on obtiendrait donc la sortie suivante.



Pour l'opérateur d'implication de Larsen, la forme globale de la fonction d'appartenance est réduite, pour la limiter au degré de vérité de la règle. Cela revient à multiplier toutes les valeurs de la fonction par le degré. Dans notre cas, on multiplie donc la fonction par 0.25, en conservant ainsi la forme de trapèze, avec les mêmes plateaux.

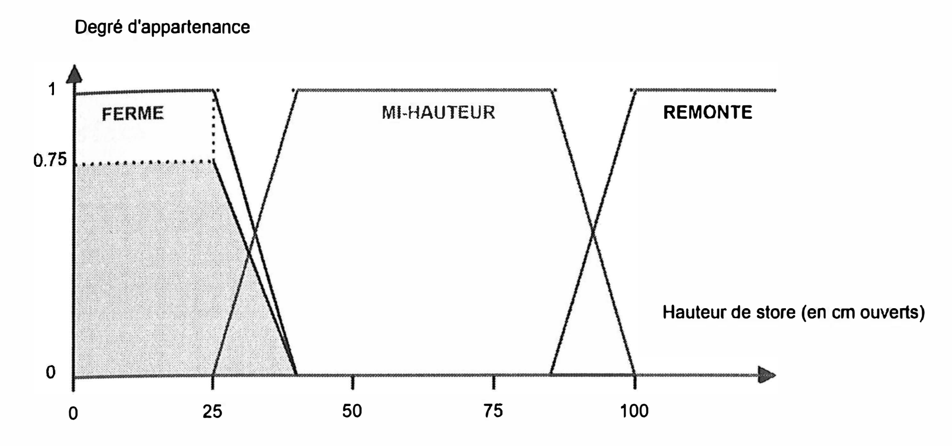


Remarque

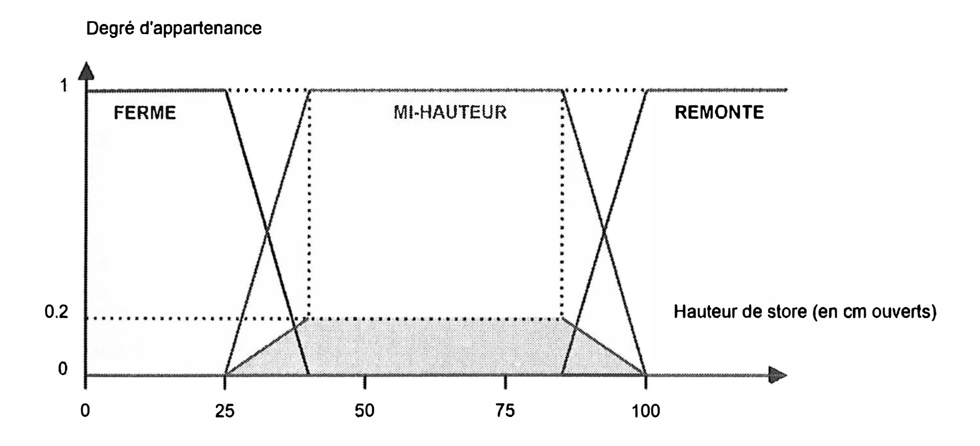
Pour la suite, nous utiliserons l'implication de Larsen. Il n'existe cependant pas une solution meilleure que l'autre dans l'absolu, cela dépend des problèmes et des préférences de chacun. L'opérateur de Larsen est ici conservé car il est plus rapide à calculer lorsqu'on travaille avec les ensembles flous, vu qu'il ne s'agit que d'une multiplication des degrés d'appartenance, alors que pour Mamdani, il faut calculer les nouvelles coordonnées des points limites du plateau.

En se basant sur une température de 21°C et un éclairage de 80000 lux, on a en réalité quatre règles qui s'appliquent : R8, R9, R11 et R12. Nous avons vu ce que l'opérateur d'implication nous donnait comme ensemble flou pour la règle R8.

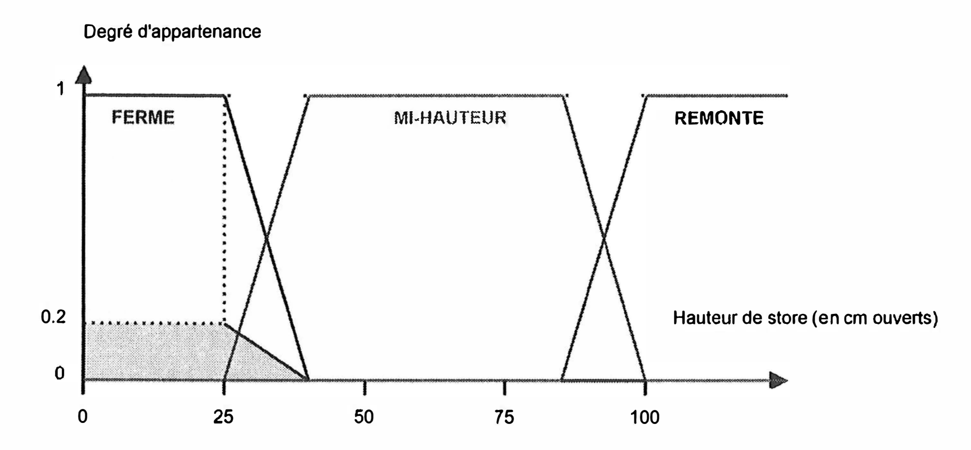
Il faut suivre le même raisonnement pour R9 : si la température est bonne (ce qui est vrai à 80 % à 21°C) et l'éclairage fort (vrai à 75 %), alors le store doit être baissé. On obtient donc l'ensemble flou suivant, de hauteur 0.75 :



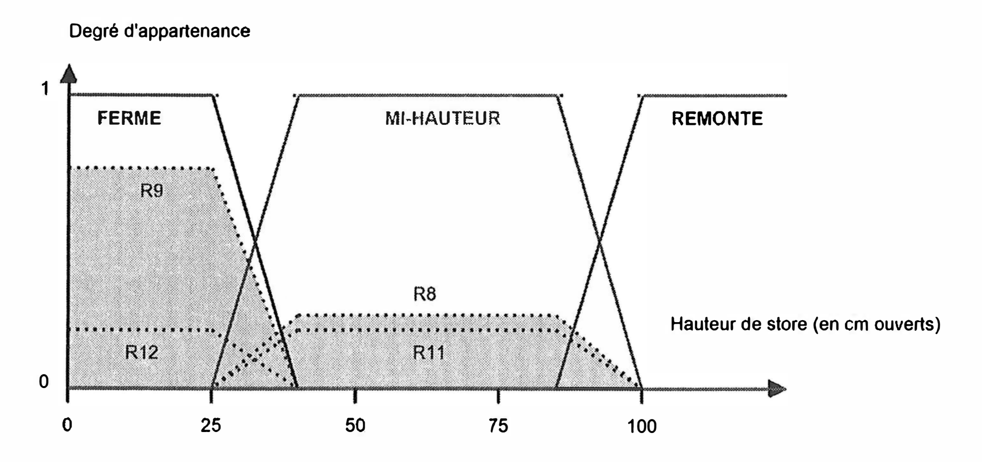
La règle R11 nous dit que si la température est chaude (vrai à 20 %) et l'éclairage moyen (vrai à 25 %), alors le store doit être à mi-hauteur (ici donc à 20 %).



Enfin, pour R12, on cherche une température chaude (vrai à 20 % à 21) et un éclairage fort (vrai à 75 %). La règle demandant un store fermé s'applique donc à 20 %.



Ces ensembles flous seront composés entre eux via l'opérateur d'union, pour obtenir la sortie floue de notre système. Nous obtenons donc l'ensemble final suivant (pour rappel, l'union consiste à prendre la hauteur maximale obtenue).



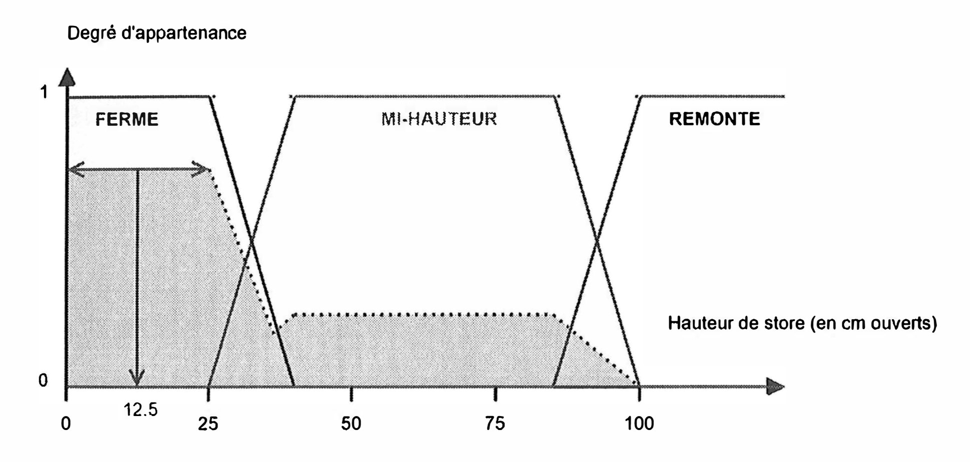
6.3 Défuzzification

Une fois l'ensemble flou résultant calculé, il faut en extraire une décision qui est une valeur numérique unique et non un ensemble flou : c'est l'étape de **défuzzification**. En effet, les actionneurs (moteurs, valves, contrôleurs, freins...) demandent un ordre qu'ils peuvent effectuer.

Dans le cas du store, il faut savoir s'il faut le monter ou le descendre. En effet, le moteur du store a besoin d'une unique valeur indiquant la hauteur à appliquer.

Là encore, il existe plusieurs solutions. Nous allons en voir deux : la défuzzification par la moyenne puis par le barycentre.

La défuzzification par la moyenne, qui est la plus simple, consiste à prendre la moyenne du plateau le plus haut. Ici, on a un plateau de 0 à 25 cm grâce à la règle R9. La moyenne est donc de 12,5 cm : le store ne laissera que 12,5 cm d'ouvert (il est donc quasiment fermé, ne laissant passer qu'un filet de lumière). Cela correspond à ce que l'on voulait : lorsqu'il fait déjà presque chaud dedans et que le soleil est fort, il est important de baisser le store.

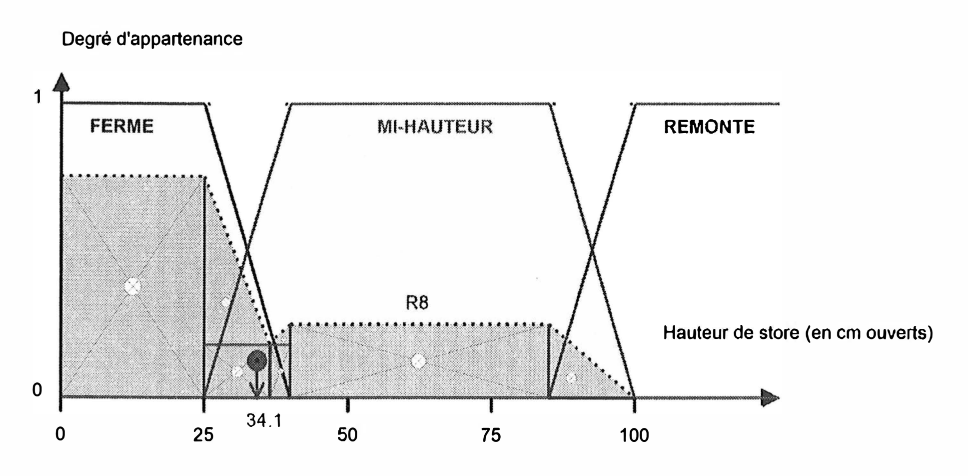


La défuzzification par le barycentre, plus complexe, consiste à chercher le barycentre (aussi appelé centroïde ou, de manière abusive, centre de gravité) de la forme obtenue. Cela permet de prendre en compte l'ensemble des règles, et non uniquement la règle majoritaire comme c'était le cas juste avant (seule la règle R9 avait participé à la décision finale).

Si on découpe la forme obtenue dans du carton, le barycentre correspond au point qui permettrait de faire tenir la forme en équilibre sur la pointe d'un stylo. Si comprendre le sens du barycentre se fait aisément, il est cependant plus compliqué à calculer. Il faut en effet calculer une moyenne pondérée des différentes formes (triangles ou rectangles) qui composent la forme globale, voire passer par des intégrales si la forme n'est pas un polygone.

Dans notre cas, on peut décomposer la forme en formes plus simples. Pour chaque sous-forme il est alors possible de trouver le barycentre : il est situé au milieu des rectangles (croisement des diagonales) et au 1/3 des triangles rectangles.

On va donc commencer par découper l'ensemble obtenu en petites formes simples (rectangles ou triangles rectangles). On associe ensuite à chaque forme son barycentre (ronds blancs). Chacun est pondéré par l'aire de la forme (graphiquement, les ronds sont plus gros dans les formes importantes). Enfin, on fait une moyenne pondérée pour obtenir le rond rouge, qui est le barycentre global de notre figure :



On obtient alors une hauteur de store de 34,1 cm, ce qui reste un store plutôt fermé, mais comme il ne fait pas si chaud que ça, on peut profiter un peu de l'éclairage.

La deuxième solution est donc bien plus précise, mais elle est plus complexe à calculer, surtout si les fonctions d'appartenance ne sont pas linéaires par morceaux. Avec les capacités actuelles des ordinateurs, elle devient cependant de plus en plus aisée et rapide, et ne présente plus de problèmes de performances.

Remarque

Il existe d'autres méthodes pour la défuzzification, moins utilisées, qui ne seront pas abordées. Elles ont cependant en commun de déterminer une unique valeur à partir d'un ensemble flou.

7. Exemples d'applications

Zadeh a posé les bases théoriques de la logique floue en 1965. Les pays occidentaux ne se sont pas vraiment intéressés à cette technique à ses débuts. Au contraire, le Japon a très vite compris son intérêt, suivi plusieurs années plus tard par le reste du monde.

7.1 Premières utilisations

Dès 1987, le premier train contrôlé par un système à base de règles floues a vu le jour à Sendai, une ville à moins de 400 km au nord de Tokyo. Les ingénieurs voulaient alors maximiser le confort des voyageurs et minimiser la consommation d'énergie alors que le véhicule devait faire de nombreux changements de vitesse.

Il s'est avéré que la consommation d'énergie a baissé de 10 % par rapport à un conducteur humain, et que les passagers vantent tous la souplesse de la conduite, principalement lors des arrêts et départs.

Il est toujours en circulation et a été le premier grand succès commercial de la logique floue.

7.2 Dans les produits électroniques

De nombreux autres constructeurs ont compris qu'un contrôleur flou pouvait améliorer le fonctionnement des machines qui nous entourent.

C'est ainsi que l'on est aujourd'hui entourés de logique floue, sans même le savoir : on en retrouve par exemple dans les lave-linge de chez LG (pour choisir le temps et la puissance idéale en fonction du contenu), les sèche-linge de Whirlpool ou les cuiseurs de riz de chez Panasonic.

7.3 En automobile

La logique floue n'est pas seulement présente dans notre électroménager, on la retrouve aussi dans nos voitures. En effet, actuellement, la grande majorité des constructeurs utilisent un contrôleur flou pour l'ABS (Antiblockiersystem, ou antiblocage des roues), Nissan et Mitsubishi en tête. Celui-ci permet de s'assurer que le freinage reste efficace, et ce quels que soient le type ou l'état de la route.

On retrouve aussi des règles floues dans les voitures à injection électronique pour décider de la quantité de carburant à utiliser, dans les boîtes automatiques pour choisir le rapport, dans certaines transmissions, dans les systèmes ESP (permettant d'éviter ou de limiter les dérapages et les pertes de contrôle) ou encore dans les régulateurs de vitesse.

7.4 Autres domaines

La logique floue est très utilisée dans les **systèmes industriels**, pour choisir l'ouverture de valves, le contrôle d'une production, la puissance des compresseurs, le fonctionnement des convertisseurs, la charge et le test de batteries... Ces contrôleurs flous permettent un gain d'énergie et/ou de durée de vie du mécanisme, par un fonctionnement plus souple, sans à-coups brusques.

La **robotique** est un autre grand domaine de cette technique, pour permettre aux robots d'avoir un comportement plus adapté et surtout plus facile à comprendre des humains qui vivent avec eux, ce qui est primordial pour des robots de compagnie.

Les programmes informatiques comme les **jeux vidéo** utilisent souvent la logique floue. C'est le cas si le comportement et le déplacement des ennemis/amis sont déterminés par des règles floues, permettant une meilleure immersion du joueur dans le jeu. *Halo*, *Thief - Deadly shadows*, *Unreal*, *Battle Cruiser: 3000AD*, *Civilization* : *Call to power*, *Close Combat* ou encore *The Sims* sont ainsi quelques grands jeux utilisant cette technique.

Enfin, la logique floue est de plus en plus utilisée dans les applications de **traitement d'images**, permettant d'améliorer les algorithmes existants, pour aider à classer des couleurs, reconnaître des formes ou extraire des informations d'images, comme l'avancement d'une maladie sur une feuille d'arbre ou dans l'agriculture raisonnée.

La logique floue est donc une technique très simple dans son principe, facile à implémenter, et qui peut trouver de très nombreux débouchés, les applications actuelles foisonnant.

8. Implémentation d'un moteur de logique floue

Cette partie indique comment coder un moteur de logique floue, en utilisant les choix préconisés précédemment. Ce code est en C#, mais il peut facilement être adapté à n'importe quel autre langage objet. Il est compatible avec le framework .NET 4 et plus, Silverlight 5 et les applications Windows ou Windows Phone Stores.

Lorsque des connaissances en mathématiques sont nécessaires, les formules utilisées sont expliquées.

8.1 Le cœur du code : les ensembles flous

8.1.1 Point2D : un point d'une fonction d'appartenance

Nous allons commencer par créer les classes de base. Pour cela, il nous faut d'abord une classe **Point2D** qui nous permet de donner les coordonnées d'un point représentatif des fonctions d'appartenance. L'axe des abscisses (x) représente la valeur numérique et l'axe des ordonnées (y) la valeur d'appartenance correspondante, entre 0 et 1.

La base de cette classe est donc la suivante :

```
using System;

public class Point2D
{
  public double X { get; set; }
  public double Y { get; set; }

  public Point2D(double x, double y)
  {
    this.X = x;
    this.Y = y;
  }
}
```

Ultérieurement, il faudra comparer des points pour connaître leur ordre. Plutôt que de devoir comparer nous-mêmes les coordonnées x des points, nous allons faire implémenter `IComparable` à cette classe. Il faut donc modifier l'en-tête pour le suivant :

```
public class Point2D : IComparable
```

Il faut ensuite ajouter la méthode `CompareTo`, qui permet de savoir si le point passé en paramètre est plus petit, égal ou plus grand que l'objet en cours (la méthode doit renvoyer respectivement un nombre positif, nul ou négatif). On se contente donc de faire la différence des abscisses :

```
  public int CompareTo(object obj)
  {
    return (int)(this.X - ((Point2D) obj).X);
  }
```

Enfin, la méthode `ToString()` permet de faciliter l'affichage :

```
public override String ToString()
{
  return "(" + this.X + ";" + this.Y + ")";
}
```

8.1.2 FuzzySet : un ensemble flou

La principale classe de notre programme, autant en termes de lignes de code que d'importance, est l'ensemble flou (ou **FuzzySet** en anglais). Elle est constituée d'une liste de points qui seront triés selon l'axe x, et de deux valeurs particulières : le minimum et le maximum que pourront prendre les valeurs numériques.

Cette classe contient donc trois propriétés :

– `Points` : la liste des points composant la fonction d'appartenance.

– `Min` : la valeur minimale possible.

– `Max` : la valeur maximale.

De plus, un constructeur est ajouté, qui initialise la liste, et deux méthodes permettant d'ajouter des points à la liste (qui est triée) : la première prend un `Point2D` en paramètre et la deuxième deux coordonnées.

Le code de base de cette classe est donc :

```
using System;
using System.Collections.Generic;
using System.Linq;

public class FuzzySet
{
  protected List<Point2D> Points;
  protected double Min { get; set; }
  protected double Max { get; set; }

  public FuzzySet(double min, double max)
  {
    this.Points = new List<Point2D>();
    this.Min = min;
    this.Max = max;
  }
```

```
    public void Add(Point2D pt)
    {
      Points.Add(pt);
      Points.Sort();
    }

    public void Add(double x, double y)
    {
      Point2D pt = new Point2D(x, y);
      Add(pt);
    }
}
```

Cette classe possède aussi une méthode `ToString()`, qui affiche l'intervalle de valeurs puis les différents points enregistrés dans la liste :

```
    public override String ToString()
    {
      String result = "[" + Min + "-" + Max + "]:";
      foreach (Point2D pt in Points)
      {
        result += pt.ToString(); // revient à "(" + pt.X + ";" +
pt.Y + ")";
      }
      return result;
    }
```

8.1.3 Opérateurs de comparaison et de multiplication

Les opérateurs de comparaison == et != permettent de savoir si deux ensembles flous sont égaux ou non. Pour cela, on s'aide de la méthode `ToString()` : deux ensembles flous sont identiques s'ils ont la même chaîne produite (étant donné que la chaîne prend en compte tous les attributs). Cela évite de comparer les points un à un.

```
    public static Boolean operator == (FuzzySet fs1, FuzzySet fs2)
    {
      return fs1.ToString().Equals(fs2.ToString());
    }

    public static Boolean operator != (FuzzySet fs1, FuzzySet fs2)
    {
      return !(fs1 == fs2);
    }
```

On ajoute enfin la multiplication par un numérique. Il faut simplement multiplier toutes les ordonnées des points par la valeur fournie et les ajouter à un nouvel ensemble flou qui sera renvoyé. Cela sera très utile lors de l'application des règles floues.

```
public static FuzzySet operator *(FuzzySet fs, double value)
{
  FuzzySet result = new FuzzySet(fs.Min, fs.Max);
  foreach (Point2D pt in fs.Points)
  {
    result.Add(new Point2D(pt.X, pt.Y * value));
  }
  return result;
}
```

8.1.4 Opérateurs ensemblistes

Le premier opérateur ensembliste et le plus simple à coder est l'opérateur **NOT**. On va créer un nouvel ensemble flou sur le même intervalle et y ajouter pour chaque point de l'ensemble de départ un point de hauteur 1-y.

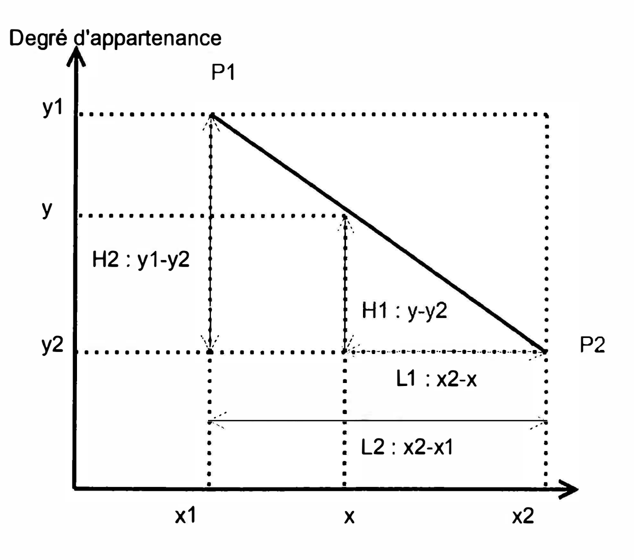
```
public static FuzzySet operator !(FuzzySet fs) {
  FuzzySet result = new FuzzySet(fs.Min, fs.Max);
  foreach (Point2D pt in fs.Points)
  {
    result.Add(new Point2D(pt.X, 1 - pt.Y));
  }
  return result;
}
```

Les méthodes restantes sont un peu plus complexes à coder. Tout d'abord, il faut une méthode qui donne le degré d'appartenance à partir d'une valeur numérique. Celle-ci est nécessaire à la fuzzification (qui consiste à transformer les valeurs numériques mesurées en degré d'appartenance pour les différentes valeurs linguistiques), mais va aussi nous servir pour l'union et l'intersection d'ensembles flous.

Pour déterminer le degré d'appartenance d'une valeur numérique, trois cas peuvent se présenter :

1. La valeur donnée est à l'extérieur de l'intervalle pour cet ensemble flou : le degré est nul.
2. Un point est défini à cette valeur dans l'ensemble flou : dans ce cas, il suffit de renvoyer le degré enregistré.
3. Aucun point n'est défini à cette valeur : il va falloir interpoler le degré d'appartenance. Pour cela, il nous faut le point immédiatement avant et le point immédiatement après.

Sur le schéma suivant ont été placés les points avant (P1) et après (P2). Ils ont donc respectivement pour coordonnées (x1, y1) et (x2, y2). Nous cherchons le degré y du point de valeur x.



Nous allons utiliser le théorème de Thalès: Comme les hauteurs H1 et H2 sont parallèles, alors on sait que :

$$\frac{L1}{L2} = \frac{H1}{H2}$$

En remplaçant chaque distance par leur expression littérale, on a :

$$\frac{x2 - x}{x2 - x1} = \frac{y - y2}{y1 - y2}$$

Soit :

$$(x2 - x) * (y1 - y2) = (y - y2) * (x2 - x1)$$

$$\frac{(y1 - y2) * (x2 - x)}{x2 - x1} = y - y2$$

On peut donc en déduire y :

$$y = \frac{(y1 - y2) * (x2 - x)}{x2 - x1} + y2$$

On obtient le code suivant :

```
public double DegreeAtValue(double Xvalue)
{
  // Cas 1 : on est en dehors de l'intervalle
  if (XValue < Min || XValue > Max)
  {
    return 0;
  }

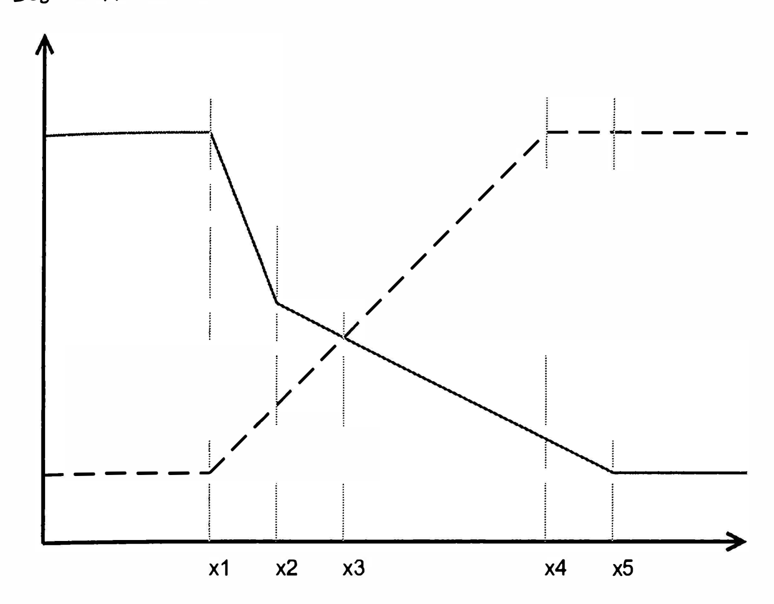
  Point2D before = Points.LastOrDefault(pt => pt.X <= Xvalue);
  Point2D after = Points.FirstOrDefault(pt => pt.X >= Xvalue);
  if (before.Equals(after))
  {
    // Cas deux : on a un point à la valeur cherchée
    return before.Y;
  }
  else
  {
    // Cas trois : on applique la formule
    return (((before.Y - after.Y) * (after.X - XValue) / (after.X
- before.X)) + after.Y);
  }
}
```

Les deux méthodes suivantes sont celles qui permettent de calculer l'intersection et l'union d'ensembles flous, c'est-à-dire les opérateurs & (ET) et | (OU). Dans les deux cas, il s'agira des opérateurs de Zadeh. Pour rappel, il s'agit d'utiliser la valeur minimale pour l'intersection et la valeur maximale pour l'union.

La difficulté se pose surtout sur le parcours des deux ensembles flous. En effet, ils ne possèdent pas forcément des points aux mêmes abscisses. Il va donc falloir parcourir les points des deux collections en parallèle. Là encore, plusieurs cas peuvent se produire :

1. Les deux ensembles possèdent un point à la même abscisse : il suffit de garder le bon degré d'appartenance (min ou max selon la méthode).
2. Un seul ensemble possède un point à une abscisse donnée : il faut calculer (grâce à la méthode précédente) le degré d'appartenance pour le deuxième ensemble et garder la bonne valeur.
3. Aucun des ensembles ne possède de point, mais les deux courbes se croisent, et donc il faut créer un point à l'intersection. La difficulté consiste surtout à détecter et calculer ces points d'intersection.

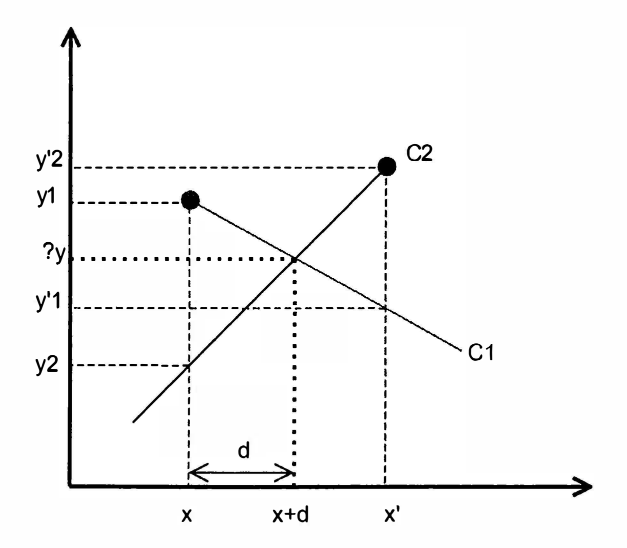
Ces cas sont illustrés dans la figure suivante :



En x1, les deux ensembles possèdent une valeur, il suffit donc de garder la bonne (min ou max selon l'opérateur). En x2, l'ensemble en trait plein possède une valeur mais pas celui en pointillés. On calcule donc son degré, et on applique l'opérateur voulu. En x3, les deux courbes s'inversent, pourtant aucune ne possède de point à cet endroit : il faut donc détecter que les fonctions se croisent pour calculer les coordonnées du point d'intersection. Enfin, en x4 et x5, un seul ensemble possède une valeur : il faut donc calculer la valeur pour le deuxième ensemble.

Il est donc nécessaire, en plus de calculer les degrés d'appartenance des ensembles pour chaque point de l'autre ensemble, de savoir quelle courbe est au-dessus de l'autre pour détecter les inversions. Il faut alors calculer les points d'intersection (qui font partie de l'ensemble résultant).

Ce calcul n'est cependant pas aisé. En effet, on ne connaît ni l'abscisse, ni l'ordonnée de ce point d'intersection, et les points avant et après dans chaque ensemble flou ne sont pas forcément alignés. On se retrouve donc dans le cas suivant :



Le point de la courbe C1, qui est le dernier observé, a pour coordonnées (x , y1). Pour la deuxième courbe (la C2), nous avons alors les coordonnées (x, y2). Le point après l'intersection a pour abscisse x', ce qui donne (x', y'1) pour la courbe C1 et (x', y'2) pour la courbe C2.

L'intersection a lieu en x+d. On sait qu'à l'intersection, les valeurs en y sont égales pour les deux courbes. Notons p1 et p2 les pentes de ces segments. On a donc :

$$y = y2 + p2 * d = y1 + p1 * d$$

On en déduit que :

$$y2 - y1 = p1 * d - p2 * d$$

$$y2 - y1 = d(p1 - p2)$$

$$\frac{y2 - y1}{p1 - p2}$$

L'intersection a donc lieu en x+d. Les pentes, elles, sont faciles à calculer et valent :

$$p1 = \frac{(y'1 - y1)}{(x' - x)} \; et \; p2 = \frac{(y'2 - y2)}{(x' - x)}$$

Nous allons donc commencer par écrire une fonction générique qui permet de fusionner deux ensembles grâce à une méthode passée en paramètre. Cette méthode attend deux valeurs et renvoie la valeur à garder :

```
  private static FuzzySet Merge(FuzzySet fs1, FuzzySet fs2,
Func<double, double, double> MergeFt)
  {
    // Code ici
  }
```

Pour l'utiliser, il suffit ensuite d'indiquer quel est l'opérateur mathématique voulu : min pour l'intersection, et max pour l'union :

```
  public static FuzzySet operator &(FuzzySet fs1, FuzzySet fs2)
  {
    return Merge(fs1, fs2, Math.Min);
  }

  public static FuzzySet operator |(FuzzySet fs1, FuzzySet fs2)
  {
    return Merge(fs1, fs2, Math.Max);
  }
```

Le code de la méthode `Merge` est le suivant :

```
  private static FuzzySet Merge(FuzzySet fs1, FuzzySet fs2,
Func<double, double, double> MergeFt)
  {
    // On crée un nouvel ensemble flou
    FuzzySet result = new FuzzySet(Math.Min(fs1.Min, fs2.Min),
Math.Max(fs1.Max, fs2.Max));

    // On va parcourir les listes via les enumérateurs
    List<Point2D>.Enumerator enum1 = fs1.Points.GetEnumerator();
```

```
    List<Point2D>.Enumerator enum2 = fs2.Points.GetEnumerator();
    enum1.MoveNext();
    enum2.MoveNext();
    Point2D oldPt1 = enum1.Current;

    // On calcule la position relative des deux courbes
    int relativePosition = 0;
    int newRelativePosition = Math.Sign(enum1.Current.Y -
enum2.Current.Y);

    // On boucle tant qu'il y a des points dans les collections
    Boolean endOfList1 = false;
    Boolean endOfList2 = false;
    while (!endOfList1 && !endOfList2)
    {
      // On récupère les valeurs x des points en cours
      double x1 = enum1.Current.X;
      double x2 = enum2.Current.X;

      // Calcul des positions relatives
      relativePosition = newRelativePosition;
      newRelativePosition = Math.Sign(enum1.Current.Y -
enum2.Current.Y);

      if (relativePosition != newRelativePosition &&
relativePosition != 0 && newRelativePosition != 0)
      {
        // Les positions ont changé :
        // on doit trouver l'intersection
        // On calcule les coordonnées des points extrêmes
        double x = (x1 == x2 ? oldPt1.X : Math.Min(x1, x2));
        double xPrime = Math.Max(x1, x2);

        // Calcul des pentes puis du delta
        double slope1 = (fs1.DegreeAtValue(xPrime) -
fs1.DegreeAtValue(x)) / (xPrime - x);
        double slope2 = (fs2.DegreeAtValue(xPrime) -
fs2.DegreeAtValue(x)) / (xPrime - x);
        double delta = (fs2.DegreeAtValue(x) -
fs1.DegreeAtValue(x)) / (slope1 - slope2);

        // On ajoute le point d'intersection
        result.Add(x + delta, fs1.DegreeAtValue(x + delta));

        // Et on passe aux points suivants
```

```
        if (x1 < x2)
        {
          oldPt1 = enum1.Current;
          endOfList1 = !(enum1.MoveNext());
        }
        else if (x1 > x2)
        {
          endOfList2 = !(enum2.MoveNext());
        }
      }
      else if (x1 == x2)
      {
        // Les deux points sont au même x, on garde le bon
        result.Add(x1, MergeFt(enum1.Current.Y, enum2.Current.Y));
        oldPt1 = enum1.Current;
        endOfList1 = !(enum1.MoveNext());
        endOfList2 = !(enum2.MoveNext());
      }
      else if (x1 < x2)
      {
        // La courbe 1 a un point avant, on calcule le degré pour
        // la deuxième courbe et on garde la bonne valeur
        result.Add(x1, MergeFt(enum1.Current.Y,
fs2.DegreeAtValue(x1)));
        oldPt1 = enum1.Current;
        endOfList1 = !(enum1.MoveNext());
      }
      else
      {
        // Ce coup-ci, c'est la courbe 2
        result.Add(x2, MergeFt(fs1.DegreeAtValue(x2),
enum2.Current.Y));
        endOfList2 = !(enum2.MoveNext());
      }
    }
    // Une des deux listes est finie, on ajoute les points restants
    if (!endOfList1)
    {
      while (!endOfList1)
      {
        result.Add(enum1.Current.X, MergeFt(0, enum1.Current.Y));
        endOfList1 = !enum1.MoveNext();
      }
    }
    else if (!endOfList2)
```

```
        {
          while (!endOfList2)
          {
            result.Add(enum2.Current.X, MergeFt(0, enum2.Current.Y));
            endOfList2 = !enum2.MoveNext();
          }
        }

        return result;
    }
}
```

8.1.5 Calcul du barycentre

La dernière méthode de cette classe est celle qui permet de déterminer le centroïde (ou barycentre) de l'ensemble, pour la défuzzification. En réalité, c'est la coordonnée en x de ce point qui est cherchée.

On peut calculer le barycentre par décomposition, puis en faisant la moyenne pondérée des coordonnées trouvées. On a alors :

$$C_x = \frac{\sum C_{ix} A_i}{\sum A_i} \text{ et } C_y = \frac{\sum C_{iy} A_i}{\sum A_i},$$

avec A_i l'aire de la forme i, et C_i les coordonnées des centroïdes de ces formes.

Il faut donc calculer pour chaque forme les coordonnées de son centroïde, et son aire. On va garder dans une variable l'aire totale, et dans une autre la somme des aires pondérées par les coordonnées. Il suffira ensuite de faire la division pour obtenir l'abscisse cherchée.

Dans notre cas, avec des fonctions d'appartenance linéaires par morceaux, il est possible de découper l'ensemble en rectangles et triangles rectangles. Les barycentres des rectangles sont situés au centre, leur coordonnée en x est donc la moyenne de leurs bords. Pour les triangles rectangles, le barycentre est situé à 1/3, du côté de l'angle droit.

En appliquant ces quelques calculs, on obtient donc la méthode suivante :

```
public double Centroid()
{
  // S'il y a moins de deux points, il n'y a pas de centroïde
  if (Points.Count < 2)
  {
    return 0;
  }
  else
  {
    // On initialise l'aire pondérée et l'aire totale
    double ponderatedArea = 0;
    double totalArea = 0;
    double localArea;
    Point2D oldPt = null;
    // On va parcourir chaque couple de points (oldPt et newPt)
    // qui délimitent une forme
    foreach (Point2D newPt in Points)
    {
      if (oldPt != null)
      {
        // Calcul du centroïde local
        if (oldPt.Y == newPt.Y)
        {
          // C'est un rectangle (même hauteur) donc au centre
          localArea = oldPt.Y * (newPt.X - oldPt.X);
          totalArea += localArea;
          ponderatedArea += ((newPt.X - oldPt.X) / 2 + oldPt.X)
* localArea;
        }
        else
        {
          // On a une forme qui est composée d'un rectangle
          // dessous et d'un triangle rectangle dessus.
          // On va faire forme par forme.
          // Pour le rectangle :
          localArea = Math.Min(oldPt.Y, newPt.Y) * (newPt.X -
oldPt.X);
          totalArea += localArea;
          ponderatedArea += ((newPt.X - oldPt.X) / 2 + oldPt.X)
* localArea;
          // Pour le triangle (centroïde à 1/3, du côté de
          // l'angle droit)
          localArea = (newPt.X - oldPt.X) * (Math.Abs(newPt.Y -
```

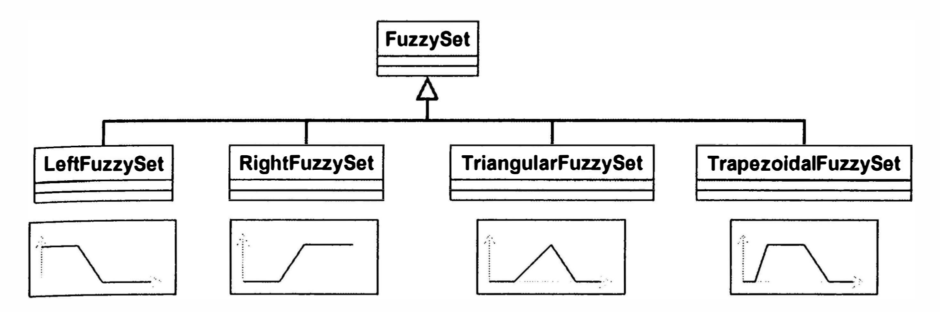
```
oldPt.Y)) / 2;
                totalArea += localArea;
                if (newPt.Y > oldPt.Y)
                {
                  ponderatedArea += (2.0 / 3.0 * (newPt.X - oldPt.X)
+ oldPt.X) * localArea;
                }
                else
                {
                  ponderatedArea += (1.0 / 3.0 * (newPt.X - oldPt.X)
+ oldPt.X) * localArea;
                }
              }
            }
            oldPt = newPt;
          }
          // On renvoie la coordonnée du centroïde qui est la somme
          // pondérée divisée par l'aire totale
          return ponderatedArea / totalArea;
        }
      }
```

Cette classe était la plus complexe à coder car elle faisait appel à plusieurs formules issues de la géométrie euclidienne. Cependant, ajouter manuellement les points peut s'avérer assez long et peu pratique.

8.2 Ensembles flous particuliers

Plusieurs classes assez simples vont être codées. Elles héritent de la classe `FuzzySet` pour faciliter la création des points appartenant à l'ensemble. Pour cela, nous allons faire quatre nouvelles classes :

- **`LeftFuzzySet`**, représentant une fonction seuil à gauche.
- **`RightFuzzySet`**, qui est une fonction seuil à droite.
- **`TriangularFuzzySet`**, qui est une fonction triangulaire.
- **`TrapezoidalFuzzySet`**, qui est une fonction trapézoïdale.



La classe **LeftFuzzySet** a besoin, en plus de son intervalle de définition, de la coordonnée du dernier point à la hauteur de 1 (heightMax) et du premier à la hauteur de 0 (baseMin). On va donc appeler le constructeur de FuzzySet puis ajouter les quatre points nécessaires :

```
public class LeftFuzzySet : FuzzySet
{
  public LeftFuzzySet(double min, double max, double heightMax,
double baseMin)
      : base(min, max)
  {
    Add(new Point2D(min, 1));
    Add(new Point2D(heightMax, 1));
    Add(new Point2D(baseMin, 0));
    Add(new Point2D(max, 0));
  }
}
```

Pour la classe **RightFuzzySet**, le principe est le même, sauf qu'il faut savoir jusqu'à quelle coordonnée le degré est nul (heightMin), puis à partir de quelle coordonnée il est de 1 (baseMax). On obtient le code suivant :

```
public class RightFuzzySet : FuzzySet
{
  public RightFuzzySet(double min, double max, double heightMin,
double baseMax)
      : base(min, max)
  {
    Add(new Point2D(min, 0));
    Add(new Point2D(heightMin, 0));
    Add(new Point2D(baseMax, 1));
    Add(new Point2D(max, 1));
  }
}
```

Pour la classe **TriangularFuzzySet**, il nous faut trois points d'intérêt : le début du triangle (triangleBegin), le point culminant (triangleCenter) et la fin du triangle (triangleEnd).

```
public class TriangularFuzzySet : FuzzySet
{
  public TriangularFuzzySet(double min, double max, double
triangleBegin, double triangleCenter, double triangleEnd) :
base(min, max)
   {
     Add(new Point2D(min, 0));
     Add(new Point2D(triangleBegin, 0));
     Add(new Point2D(triangleCenter, 1));
     Add(new Point2D(triangleEnd, 0));
     Add(new Point2D(max, 0));
   }
}
```

Enfin, pour la classe **TrapezoidalFuzzySet**, il nous faut quatre points : l'extrémité gauche de la base (baseLeft), puis l'extrémité gauche du plateau (heightLeft), l'extrémité droite du plateau (heightRight) et enfin l'extrémité droite de la base (baseRight).

```
public class TrapezoidalFuzzySet : FuzzySet
{
  public TrapezoidalFuzzySet(double min, double max, double
baseLeft, double heightLeft, double heightRight, double baseRight)
    : base(min, max)
   {
     Add(new Point2D(min, 0));
     Add(new Point2D(baseLeft, 0));
     Add(new Point2D(heightLeft, 1));
     Add(new Point2D(heightRight, 1));
     Add(new Point2D(baseRight, 0));
     Add(new Point2D(max, 0));
   }
}
```

Ces quatre classes ne sont pas obligatoires, mais elles vont permettre de simplifier la création d'ensembles flous.

8.3 Variables et valeurs linguistiques

Les ensembles flous étant définis, nous allons pouvoir créer les valeurs linguistiques, puis les variables linguistiques.

8.3.1 LinguisticValue : valeur linguistique

Une valeur linguistique (**`LinguisticValue`**) est simplement un nom associé à un ensemble flou, par exemple "Chaud" et l'ensemble flou qui le caractérise.

La classe ne possède donc que deux propriétés : `Fs` qui est un ensemble flou (`FuzzySet`) et `Name`, de type `String` (le nom de la valeur).

Elle possède aussi deux méthodes :

– Un constructeur permettant l'initialisation de ses attributs.
– Une méthode renvoyant le degré d'appartenance d'une valeur numérique donnée (cette méthode se contente d'appeler la méthode de l'ensemble flou codée précédemment).

Voici donc le code de cette classe :

```
using System;

public class LinguisticValue
{
  internal FuzzySet Fs { get; set; }
  internal String Name { get; set; }

  public LinguisticValue(String _name, FuzzySet _fs)
  {
    Name = _name;
    Fs = _fs;
  }

  internal double DegreeAtValue(double val)
  {
    return Fs.DegreeAtValue(val);
  }
}
```

8.3.2 LinguisticVariable : variable linguistique

La classe variable linguistique est simple elle aussi. Une variable linguistique, c'est tout d'abord un nom (par exemple "Température"), puis une plage de valeurs (valeurs min et max) et enfin une liste de valeurs linguistiques. Elle possède donc quatre propriétés : `Name` (de type `String`), `MinValue` et `MaxValue` (les valeurs limites) et `Values`, une liste de `LinguisticValue`.

En plus d'un constructeur, on propose trois méthodes. La première, `AddValue` ajoute une nouvelle valeur linguistique : soit en fournissant un objet `LinguisticValue`, soit en le créant à partir d'un nom et d'un ensemble flou. La deuxième permet d'effacer la liste des valeurs enregistrées (`ClearValues`). La dernière permet de retrouver une valeur à partir de son nom (`LinguisticValueByName`). Ces deux dernières utilisent les fonctions sur les listes existantes dans le framework.

```
using System;
using System.Collections.Generic;

public class LinguisticVariable
{
  internal String Name { get; set; }
  List<LinguisticValue> Values { get; set; }
  internal Double MinValue { get; set; }
  internal Double MaxValue { get; set; }

  public LinguisticVariable(String _name, double _min, double_max)
  {
    Values = new List<LinguisticValue>();
    Name = _name;
    MinValue = _min;
    MaxValue = _max;
  }

  public void AddValue(LinguisticValue lv) {
    Values.Add(lv);
  }

  public void AddValue(String name, FuzzySet fs)
  {
    Values.Add(new LinguisticValue(name, fs));
  }
```

```
  public void ClearValues() {
    Values.Clear();
  }

  internal LinguisticValue LinguisticValueByName(string name)
  {
    name = name.ToUpper();
    foreach (LinguisticValue val in Values)
    {
      if (val.Name.ToUpper().Equals(name))
      {
        return val;
      }
    }
    return null;
  }
}
```

8.4 Règles floues

Les règles floues s'expriment sous la forme :

```
IF Variable IS Valeur AND ... THEN Variable IS Valeur
```

Par exemple :

```
IF Température IS Chaude AND Ensoleillement IS Fort THEN Store IS Bas
```

8.4.1 FuzzyExpression : expression floue

On commence par définir une expression floue (**`FuzzyExpression`**) pour exprimer la forme "Variable IS Valeur". Pour cela, nous allons donc associer une variable linguistique `Lv` et une chaîne correspondant à la valeur linguistique voulue (`LinguisticValueName`). Par exemple, pour "Température IS Chaude", `Lv` serait la variable linguistique Température, et `LinguisticValueName` serait la chaîne "Chaude".

Le code de cette classe est donc le suivant :

```
using System;
public class FuzzyExpression
{
  internal LinguisticVariable Lv { get; set; }
```

```
    internal String LinguisticValueName { get; set; }

    public FuzzyExpression(LinguisticVariable _lv, String _value)
    {
      Lv = _lv;
      LinguisticValueName = _value;
    }
}
```

8.4.2 FuzzyValue : valeur floue

Nous allons aussi définir une "valeur floue" (**FuzzyValue**), qui nous permettra de pouvoir indiquer ensuite à nos règles qu'actuellement la température est de 21°, par exemple. Il faut donc associer une variable linguistique et sa valeur numérique correspondante.

Le code de cette classe est très simple :

```
public class FuzzyValue
{
  internal LinguisticVariable Lv;
  internal double Value;

  public FuzzyValue(LinguisticVariable _lv, double _value)
  {
    Lv = _lv;
    Value = _value;
  }
}
```

8.4.3 FuzzyRule : règle floue

Une fois ces deux classes écrites, nous allons pouvoir définir les règles floues (**FuzzyRule**). Celles-ci contiennent une liste d'expressions floues en prémisses (la partie avant le "THEN") et une expression floue en conclusion. La base de cette classe est donc la suivante (nous allons ensuite la compléter) :

```
using System;
using System.Collections.Generic;

public class FuzzyRule
{
  List<FuzzyExpression> Premises;
  FuzzyExpression Conclusion;
```

```
  public FuzzyRule(List<FuzzyExpression> _prem, FuzzyExpression_concl)
  {
    Premises = _prem;
    Conclusion = _concl;
  }
}
```

La méthode `Apply` permet d'appliquer une liste de `FuzzyValue` (qui est donc la définition du cas à résoudre) à la règle en cours. Cela produit un ensemble flou. Tous les ensembles flous de toutes les règles seront ensuite cumulés pour obtenir la sortie floue, qui subira la défuzzification.

Pour cela, on parcourt chaque prémisse de la règle et, à chaque fois, on cherche quelle est la valeur numérique correspondant à la valeur linguistique (par exemple, on cherche "21°" pour la valeur "chaude"). Une fois trouvée, on calcule le degré d'appartenance. On répète cette recherche sur toutes les prémisses, en gardant à chaque fois la valeur minimale obtenue. En effet, le degré d'application d'une règle correspond au plus petit degré d'appartenance des prémisses.

Une fois le degré obtenu, on calcule l'ensemble flou résultant, qui est la multiplication de l'ensemble par le degré (implication de Larsen).

Le code est donc le suivant :

```
  internal FuzzySet Apply(List<FuzzyValue> Problem)
  {
    double degree = 1;
    foreach (FuzzyExpression premise in Premises)
    {
      double localDegree = 0;
      LinguisticValue val = null;
      foreach (FuzzyValue pb in Problem)
      {
        if (premise.Lv == pb.Lv)
        {
          val =
premise.Lv.LinguisticValueByName(premise.LinguisticValueName);
          if (val != null)
          {
            localDegree = val.DegreeAtValue(pb.Value);
            break;
          }
```

```
            }
          }
          if (val == null)
          {
            return null;
          }

          degree = Math.Min(degree, localDegree);
        }
        return Conclusion.Lv.LinguisticValueByName(Conclusion.Linguistic
ValueName).Fs * degree;
      }
```

Une nouvelle méthode sera ajoutée ultérieurement à cette classe pour simplifier l'écriture des règles. Cependant, nous allons d'abord définir le contrôleur général.

8.5 Système de contrôle flou

Pour gérer tout notre système, un système flou (**FuzzySystem**) est implémenté. Celui-ci permet de gérer les différentes variables linguistiques en entrée, la variable linguistique de sortie, les différentes règles, et le problème à résoudre.

Notre version de base contient donc cinq propriétés :

- `Name` : le nom du système.
- `Inputs` : la liste des variables linguistiques en entrée.
- `Output` : la variable linguistique en sortie.
- `Rules` : la liste des règles floues à appliquer.
- `Problem` : la liste des valeurs numériques du problème à résoudre.

On lui ajoute un constructeur et les méthodes permettant d'ajouter les variables linguistiques (addInputVariable et addOutputVariable) ou les règles (addFuzzyRule). Deux méthodes permettent de créer un nouveau cas à résoudre par insertion de nouvelles valeurs (SetInputVariable) ou remise à zéro (ResetCase). Enfin, on ajoute une méthode permettant de retrouver une variable linguistique à partir de son nom (LinguisticVariableByName).

Le code est donc le suivant :

```
using System;
using System.Collections.Generic;

public class FuzzySystem
{
  String Name { get; set; }
  List<LinguisticVariable> Inputs;
  LinguisticVariable Output;
  List<FuzzyRule> Rules;
  List<FuzzyValue> Problem;

  // Constructeur
  public FuzzySystem(String _name)
  {
    Name = _name;
    Inputs = new List<LinguisticVariable>();
    Rules = new List<FuzzyRule>();
    Problem = new List<FuzzyValue>();
  }

  // Ajout d'une variable linguistique en entrée
  public void addInputVariable(LinguisticVariable lv)
  {
    Inputs.Add(lv);
  }

  // Ajout d'une variable linguistique en sortie
  public void addOutputVariable(LinguisticVariable lv)
  {
    Output = lv;
  }

  // Ajout d'une règle
  public void addFuzzyRule(FuzzyRule fuzzyRule)
```

```
    {
      Rules.Add(fuzzyRule);
    }

    // Ajout d'une valeur numérique en entrée
    public void SetInputVariable(LinguisticVariable inputVar,
double value)
    {
      Problem.Add(new FuzzyValue(inputVar, value));
    }

    // Remise à zéro des valeurs en entrée pour changer de cas
    public void ResetCase()
    {
      Problem.Clear();
    }

    // retrouver une variable linguistique à partir de son nom
    internal LinguisticVariable LinguisticVariableByName(string name)
    {
      foreach (LinguisticVariable input in Inputs)
      {
        if (input.Name.ToUpper().Equals(name))
        {
          return input;
        }
      }
      if (Output.Name.ToUpper().Equals(name))
      {
        return Output;
      }
      return null;
    }
}
```

C'est dans cette classe que l'on trouve la méthode principale permettant de résoudre un problème flou et de renvoyer la valeur numérique attendue : `Solve()`. Celle-ci suit les étapes vues précédemment, à savoir l'application des règles, une par une, puis la défuzzification de l'ensemble flou résultant.

```
    public double Solve()
    {
      // Initialisation du résultat
      FuzzySet res = new FuzzySet(Output.MinValue, Output.MaxValue);
      res.Add(Output.MinValue, 0);
```

```
    res.Add(Output.MaxValue, 0);

    // Application des règles et calcul
    // du fuzzy set résultant (union)
    foreach (FuzzyRule rule in Rules)
    {
      res = res | rule.Apply(Problem);
    }

    // Défuzzification
    return res.Centroid();
  }
```

Il n'est par contre pas très facile de donner les règles via les expressions floues. Il est plus simple d'utiliser des règles sous leur forme textuelle. On ajoute une méthode qui permettra de créer une règle non pas à partir d'un objet `FuzzyRule` mais à partir d'une chaîne de caractères.

```
  public void addFuzzyRule(string ruleStr)
  {
    FuzzyRule rule = new FuzzyRule(ruleStr, this);
    Rules.Add(rule);
  }
```

Cette méthode utilise un nouveau constructeur de règle (à ajouter dans la classe **`FuzzyRule`**) qui décompose la règle écrite en prémisses et conclusion, puis qui décompose chaque partie en expression floue. Cette méthode utilise de manière importante les fonctions de manipulation de chaînes, en particulier `Split` (pour couper une chaîne en sous-chaînes), `ToUpper` (pour mettre une chaîne en majuscules) et `Remove` (pour enlever un certain nombre de caractères).

Le code commenté est le suivant :

```
  public FuzzyRule(string ruleStr, FuzzySystem fuzzySystem)
  {
    // On met la chaîne en majuscules
    ruleStr = ruleStr.ToUpper();

    // On sépare les prémisses de la conclusion
    // par la présence du mot-clé THEN
    String[] rule = ruleStr.Split(new String[]{" THEN "},
StringSplitOptions.RemoveEmptyEntries);
    if (rule.Length == 2)
```

```
    {
      // On a 2 parties, donc la syntaxe actuelle est exacte
      rule[0] = rule[0].Remove(0, 2); // On enlève "IF" du début
      // On va maintenant séparer et traiter les prémisses (AND)
      String[] prem = rule[0].Trim().Split(new String[] {" AND "},
StringSplitOptions.RemoveEmptyEntries);
      Premises = new List<FuzzyExpression>();
      foreach (String exp in prem)
      {
        // On coupe chaque expression avec le mot-clé IS
        // et on crée la FuzzyExpression correspondante
        // qu'on ajoute aux prémisses
        String[] res = exp.Split(new String[] { " IS " },
StringSplitOptions.RemoveEmptyEntries);
        if (res.Length == 2)
        {
          FuzzyExpression fexp = new
FuzzyExpression(fuzzySystem.LinguisticVariableByName(res[0]),
res[1]);
          Premises.Add(fexp);
        }
      }
      // On traite de la même façon la conclusion
      String[] conclu = rule[1].Split(new String[] {" IS "},
StringSplitOptions.RemoveEmptyEntries);
      if (conclu.Length == 2)
      {
        Conclusion = new
FuzzyExpression(fuzzySystem.LinguisticVariableByName(conclu[0]),
conclu[1]);
      }
    }
  }
```

8.6 Synthèse du code créé

Notre système est maintenant complet. Nous avons donc dû créer plusieurs classes :

- **Point2D** qui sert d'utilitaire pour les ensembles flous.
- **FuzzySet** qui représente l'ensemble flou et possède les opérateurs d'union, d'intersection, de multiplication, de comparaison et le calcul du centroïde.

- Quatre classes héritant de **FuzzySet** pour simplifier la création de nouveaux ensembles flous.
- **LinguisticVariable** et **LinguisticValue** qui permettent de définir des variables et des valeurs floues.
- **FuzzyExpression** et **FuzzyValue**, qui permettent de définir des parties de règles ou un cas à traiter.
- **FuzzyRule** qui gère les règles, et les crée à partir d'une chaîne de caractères. C'est aussi cette classe qui permet de créer l'ensemble flou résultant.
- **FuzzySystem** qui gère l'ensemble.

9. Implémentation d'un cas pratique

Nous allons utiliser la logique floue pour contrôler un GPS de voiture, plus précisément le niveau de zoom. En effet, en fonction de la distance au prochain changement de direction et de la vitesse à laquelle on roule, le niveau de zoom utilisé n'est pas le même : lorsqu'on se rapproche d'un changement de direction ou que l'on ralentit, le zoom augmente pour nous montrer de plus en plus de détails.

Pour avoir un rendu fluide et non saccadé, un contrôleur flou est donc utilisé. Pour cela, on commence par créer une nouvelle classe contenant juste une méthode `main` pour le moment (qui boucle, pour permettre de conserver les affichages ensuite) :

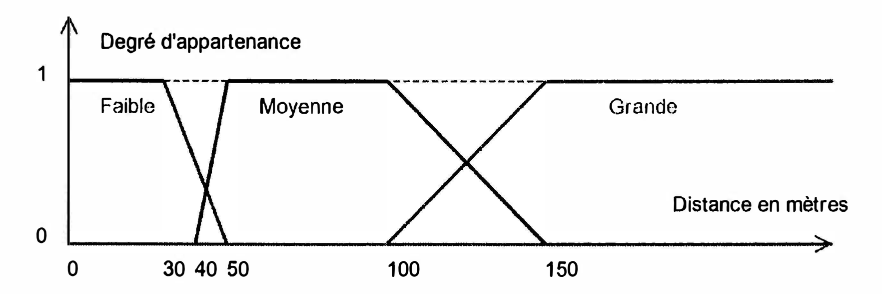
```
using System;

public class ZoomGPS
{
  static void Main(string[] args)
  {
    // Le code sera placé ici
    while (true) ;
  }
}
```

On commence par créer un nouveau contrôleur flou dans le `main` :

```
// Création du système
FuzzySystem system = new FuzzySystem("Gestion du zoom GPS");
```

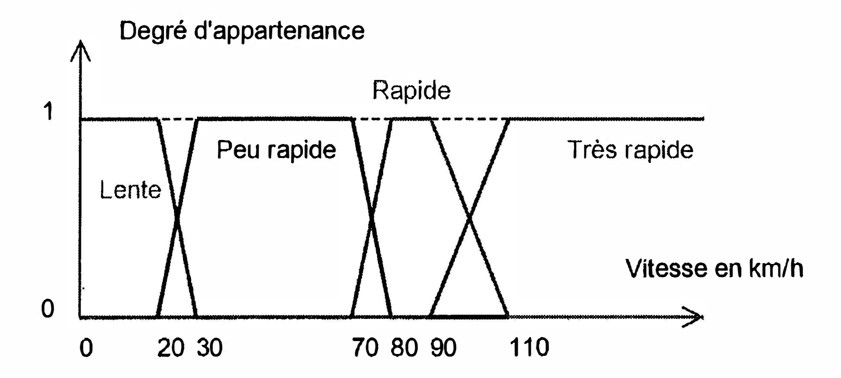
L'étape suivante consiste à définir les différentes variables linguistiques. Nous en aurons trois : Distance et Vitesse en entrée, et Zoom en sortie. Pour la distance (en mètres jusqu'au prochain changement de direction), on va créer trois variables linguistiques : "faible", "moyenne" et "grande".
Voici le schéma reprenant ces ensembles flous :



Au niveau du code, on va donc créer la variable linguistique, puis lui ajouter les trois valeurs linguistiques, et enfin ajouter cette variable comme entrée au système :

```
    Console.WriteLine("1) Ajout des variables");
    // Ajout de la variable linguistique "Distance" (de 0 à 500 000 m)
    LinguisticVariable distance = new LinguisticVariable("Distance",
0, 500000);
    distance.AddValue(new LinguisticValue("Faible", new
LeftFuzzySet(0, 500000, 30, 50)));
    distance.AddValue(new LinguisticValue("Moyenne", new
TrapezoidalFuzzySet(0, 500000, 40, 50, 100, 150)));
    distance.AddValue(new LinguisticValue("Grande", new
RightFuzzySet(0, 500000, 100, 150)));
    system.addInputVariable(distance);
```

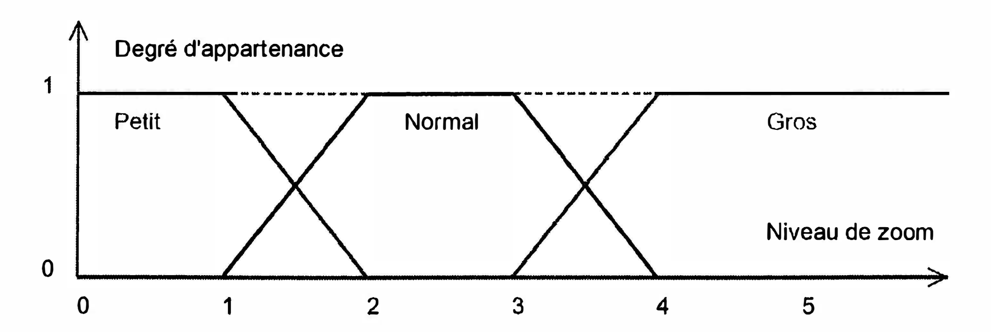
On procède de même avec la vitesse. Voici les différentes valeurs et les ensembles flous correspondants :



On a donc le code suivant :

```
    // Ajout de la variable linguistique "Vitesse" (de 0 à 200)
    LinguisticVariable vitesse = new
LinguisticVariable("Vitesse", 0, 200);
    vitesse.AddValue(new LinguisticValue("Lente", new
LeftFuzzySet(0, 200, 20, 30)));
    vitesse.AddValue(new LinguisticValue("PeuRapide", new
TrapezoidalFuzzySet(0, 200, 20, 30, 70, 80)));
    vitesse.AddValue(new LinguisticValue("Rapide", new
TrapezoidalFuzzySet(0, 200, 70, 80, 90, 110)));
    vitesse.AddValue(new LinguisticValue("TresRapide", new
RightFuzzySet(0, 200, 90, 110)));
    system.addInputVariable(vitesse);
```

Enfin, pour le niveau de zoom, on définit trois valeurs linguistiques. Le niveau est défini par une valeur numérique entre 0 (la carte est dézoomée, on voit donc loin) et 5 (niveau de zoom maximal, avec beaucoup de détails mais une visibilité faible).



Le code de création de cette variable (en sortie du système) est le suivant :

```
    // Ajout de la variable linguistique "Zoom" (de 1 à 5)
    LinguisticVariable zoom = new LinguisticVariable("Zoom", 0, 5);
    zoom.AddValue(new LinguisticValue("Petit", new
LeftFuzzySet(0, 5, 1, 2)));
    zoom.AddValue(new LinguisticValue("Normal", new
TrapezoidalFuzzySet(0, 5, 1, 2, 3, 4)));
    zoom.AddValue(new LinguisticValue("Gros", new
RightFuzzySet(0, 5, 3, 4)));
    system.addOutputVariable(zoom);
```

Une fois les variables créées, il va falloir créer les règles. On décide d'appliquer les règles suivantes (elles indiquent le niveau de zoom en fonction de la vitesse et de la distance). Par exemple, si la vitesse est lente et la distance grande, alors le zoom doit être petit.

Distance → **Vitesse ↓**	**Faible**	**Moyenne**	**Grande**
Lente	Normal	Petit	Petit
Peu rapide	Normal	Normal	Petit
Rapide	Gros	Normal	Petit
Très rapide	Gros	Gros	Petit

On code donc ces différentes règles. Pour gagner un peu de temps, comme le zoom doit être petit si la distance est faible et ce, quelle que soit la distance, on regroupe tous ces cas en une seule règle. On a donc 9 règles pour couvrir les 12 cas possibles.

```
    Console.WriteLine("2) Ajout des règles");

    system.addFuzzyRule("IF Distance IS Grande THEN Zoom IS 
Petit");
    system.addFuzzyRule("IF Distance IS Faible AND Vitesse IS 
Lente THEN Zoom IS Normal");
    system.addFuzzyRule("IF Distance IS Faible AND Vitesse IS 
PeuRapide THEN Zoom IS Normal");
    system.addFuzzyRule("IF Distance IS Faible AND Vitesse IS 
Rapide THEN Zoom IS Gros");
    system.addFuzzyRule("IF Distance IS Faible AND Vitesse IS 
TresRapide THEN Zoom IS Gros");
    system.addFuzzyRule("IF Distance IS Moyenne AND Vitesse IS 
Lente THEN Zoom IS Petit");
    system.addFuzzyRule("IF Distance IS Moyenne AND Vitesse IS 
PeuRapide THEN Zoom IS Normal");
    system.addFuzzyRule("IF Distance IS Moyenne AND Vitesse IS 
Rapide THEN Zoom IS Normal");
    system.addFuzzyRule("IF Distance IS Moyenne AND Vitesse IS 
TresRapide THEN Zoom IS Gros");
    Console.WriteLine("9 règles ajoutées \n");
```

Nous allons maintenant vouloir résoudre cinq cas différents, résumés dans le tableau ci-dessous :

Cas n°	**1**	**2**	**3**	**4**	**5**
Distance	70 m	70 m	40 m	110 m	160 m
Vitesse	35 km/h	25 km/h	72,5 km/h	100 km/h	45 km/h

Nous allons donc coder ces différents cas :

```
    Console.WriteLine("3) Résolution de cas pratiques");
    // Cas pratique 1 : vitesse de 35 km/h,
    // et prochain changement de direction à 70 m
    Console.WriteLine("Cas 1 :");
    system.SetInputVariable(vitesse, 35);
    system.SetInputVariable(distance, 70);
    Console.WriteLine("Résultat : " + system.Solve() + "\n");

    // Cas pratique 2 : vitesse de 25 km/h,
    // et prochain changement de direction à 70 m
    system.ResetCase();
    Console.WriteLine("Cas 2 :");
```

```
    system.SetInputVariable(vitesse, 25);
    system.SetInputVariable(distance, 70);
    Console.WriteLine("Résultat : " + system.Solve() + "\n");

    // Cas pratique 3 : vitesse de 72.5 km/h,
    // et prochain changement de direction à 40 m
    system.ResetCase();
    Console.WriteLine("Cas 3 :");
    system.SetInputVariable(vitesse, 72.5);
    system.SetInputVariable(distance, 40);
    Console.WriteLine("Résultat : " + system.Solve() + "\n");

    // Cas pratique 4 : vitesse de 100 km/h,
    // et prochain changement de direction à 110 m
    system.ResetCase();
    Console.WriteLine("Cas 4 :");
    system.SetInputVariable(vitesse, 100);
    system.SetInputVariable(distance, 110);
    Console.WriteLine("Résultat : " + system.Solve() + "\n");

    // Cas pratique 5 : vitesse de 45 km/h, et changement à 160 m
    system.ResetCase();
    Console.WriteLine("Cas 5 :");
    system.SetInputVariable(vitesse, 45);
    system.SetInputVariable(distance, 160);
    Console.WriteLine("Résultat : " + system.Solve() + "\n");
```

Voici le résultat obtenu en lançant ce programme :

```
1) Ajout des variables
2) Ajout des règles
9 règles ajoutées

3) Résolution de cas pratiques
Cas 1 :
Résultat : 2,5

Cas 2 :
Résultat : 1,78205128205128

Cas 3 :
Résultat : 2,93189964157706

Cas 4 :
Résultat : 2,89196256537297
```

```
Cas 5 :
Résultat : 0,777777777777778
```

On voit ainsi que le niveau de zoom, selon les cas, ira de 0.78 pour le cinquième cas à 2.93 pour le troisième cas. Cela vérifie bien le comportement attendu : plus le prochain changement de direction est prêt ou plus on va lentement et plus il est important de voir les détails dans le GPS (donc on souhaite un niveau de zoom plus fort).

10. Synthèse

La logique floue permet de prendre des décisions en fonction de règles imprécises, c'est-à-dire dont l'évaluation est soumise à interprétation.

Pour cela, on définit des variables linguistiques (comme la température) et on leur associe des valeurs linguistiques ("chaud", "froid"...). À chaque valeur, on fait correspondre un ensemble flou, déterminé par sa fonction d'appartenance : pour toutes les valeurs numériques possibles, elle associe un degré d'appartenance entre 0 (la valeur linguistique est totalement fausse) et 1 (elle est totalement vraie), en passant par des stades intermédiaires.

Une fois les variables et valeurs linguistiques déterminées, les règles à appliquer sont indiquées au système flou. On lui donne ensuite les valeurs numériques mesurées.

La première étape pour fournir une décision est la fuzzification qui consiste à associer à chaque valeur numérique son degré d'appartenance aux différentes valeurs linguistiques. Les règles peuvent alors être appliquées, et la somme des règles (constituée par l'union des ensembles flous résultants) fournit un nouvel ensemble flou.

La défuzzification est l'étape permettant de passer de cet ensemble flou à la valeur numérique représentant la décision. Plusieurs méthodes sont possibles mais le calcul du barycentre reste la plus précise, pour une charge de calculs supplémentaires faible au vu de la capacité actuelle des ordinateurs.

Il ne reste alors plus qu'à appliquer cette décision. Le système de gestion flou permet ainsi un contrôle en temps réel, et applique de petites modifications dans la sortie si les entrées bougent légèrement, ce qui permet plus de souplesse, et une plus faible usure des systèmes mécaniques contrôlés. De plus, cela est plus proche du comportement d'un humain, ce qui permet de multiples utilisations.

Chapitre 3
Recherche de chemins

1. Présentation du chapitre

De nombreux domaines font face à un problème de recherche de chemins, appelé "pathfinding" en anglais. On pense tout d'abord aux GPS et aux logiciels de recherche d'itinéraires (en voiture, en train, en transport en commun...), voire aux jeux vidéo dans lesquels les ennemis doivent arriver sur le joueur par le chemin le plus court.

La recherche de chemins est en réalité un domaine bien plus vaste. En effet, de nombreux problèmes peuvent être représentés sous la forme d'un graphe, comme l'enchaînement des mouvements dans un jeu d'échecs.

La recherche d'un chemin dans ce cas-là peut être vue comme la recherche de la suite des mouvements à faire pour gagner.

Ce chapitre commence par présenter les différents concepts de théorie des graphes, et les définitions associées. Les algorithmes fondamentaux sont ensuite présentés, avec leur fonctionnement et leurs contraintes.

Les principaux domaines dans lesquels on peut utiliser cette recherche de chemins sont alors indiqués et un exemple d'implémentation des algorithmes en C# est présenté et appliqué à une recherche de chemins dans un environnement en 2D.

Le chapitre se termine par une synthèse.

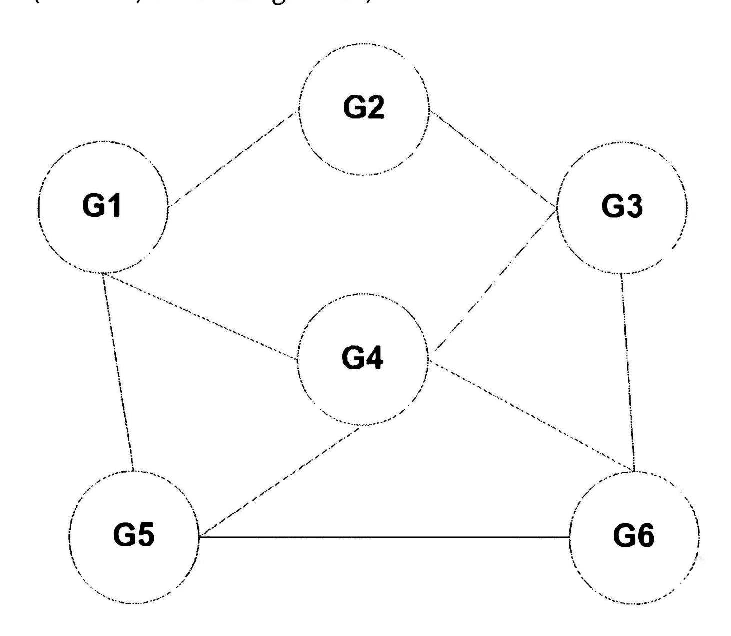
2. Chemins et graphes

Un chemin peut être vu comme un parcours dans un graphe. Les principaux algorithmes se basent donc sur la **théorie des graphes**.

2.1 Définition et concepts

Un **graphe** est un ensemble de **nœuds** ou **sommets** (qui peuvent représenter par exemple des villes) liés par des **arcs**, qui seraient alors des routes.

Voici un graphe qui représente des gares et les liens qui existent entre ces gares (en train, sans changement) :



Les gares de G1 à G6 sont donc les nœuds. L'arc allant de G5 à G6 indique la présence d'un lien direct entre ces deux gares. Il est noté (G5, G6) ou (G6, G5) selon le sens voulu.

Par contre pour aller de G1 à G6, il n'y a pas de lien direct, il faudra passer par G4 ou G5 si on ne souhaite qu'un changement, ou par G2 puis G3 avec deux changements.

Un **chemin** permet de rejoindre différents sommets liés entre eux par des arcs. Ainsi, G1-G2-G3-G6 est un chemin de **longueur** 3 (la longueur est le nombre d'arcs suivis).

On parle de **circuit** lorsqu'on peut partir d'un nœud et y revenir. Ici, le graphe contient de nombreux circuits, comme G1-G4-G5-G1 ou G4-G5-G6-G4.

L'**ordre** d'un graphe correspond au nombre de sommets qu'il contient. Notre exemple contient 6 gares, il s'agit donc d'un graphe d'ordre 6.

Deux nœuds sont dits **adjacents** (ou voisins) s'il existe un lien permettant d'aller de l'un à l'autre. G5 est donc adjacent à G1, G4 et G6.

2.2 Représentations

2.2.1 Représentation graphique

Il existe plusieurs façons de représenter un graphe. La première est la **représentation graphique**, comme celle vue précédemment.

L'ordre et le placement des nœuds ne sont pas importants, cependant on va chercher à toujours placer les sommets de façon à rendre le graphe le plus lisible possible.

Le graphe est dit **orienté** si les arcs ont un sens, représentant par exemple des rues à sens unique dans une ville. Si tous les arcs peuvent être pris dans les deux sens, on dit alors que le graphe est **non orienté**, ce qui est généralement le cas de ceux utilisés pour la recherche de chemins.

2.2.2 Matrice d'adjacence

Les représentations graphiques ne sont pas toujours très pratiques, en particulier quand il s'agit d'y appliquer des algorithmes ou de les rentrer dans un ordinateur.

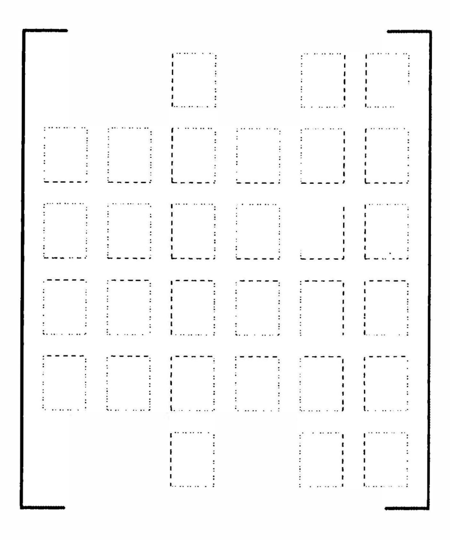
On préfère souvent utiliser une matrice, appelée **matrice d'adjacence**.

Remarque

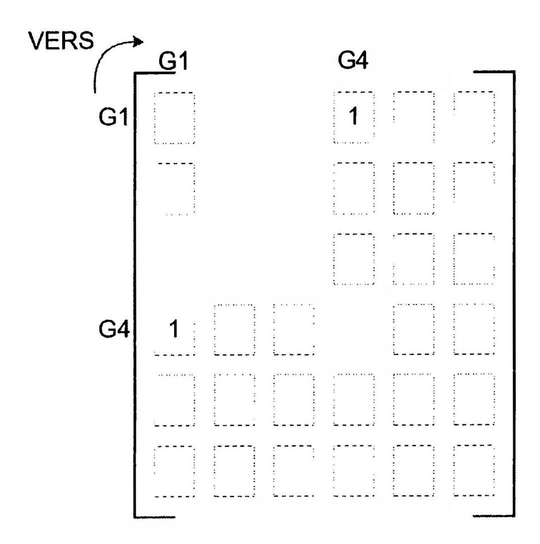
Une matrice est une structure mathématique particulière qui peut être vue plus simplement comme un tableau à deux dimensions.

Dans cette matrice, l'absence d'arc est représentée par un 0, et sa présence par un 1.

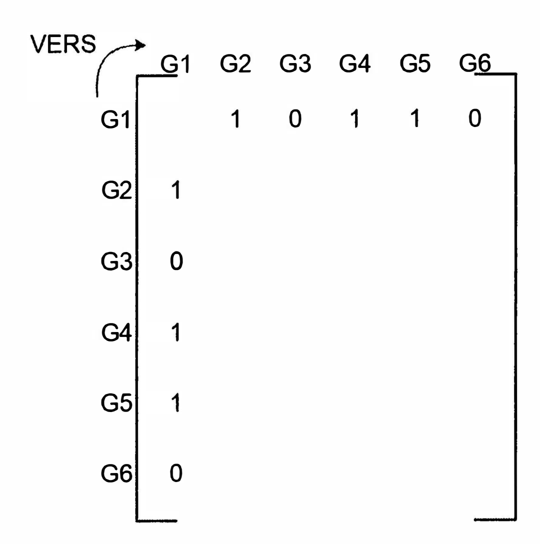
Dans l'exemple des gares, on a donc une matrice de 6 par 6 (car il y a 6 gares) :



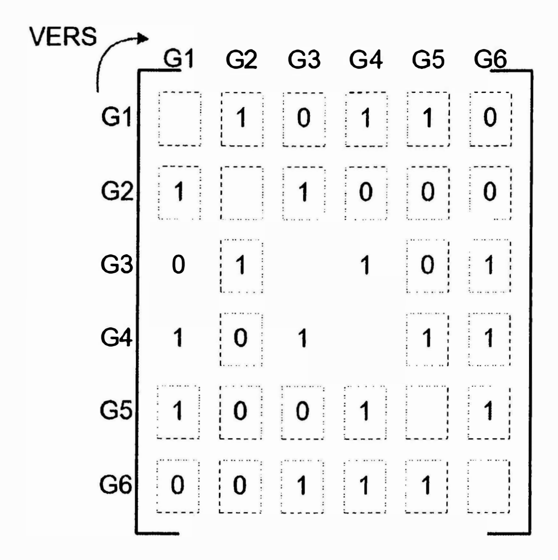
On voit sur le graphe qu'il existe un lien entre G1 et G4. La case correspondant au trajet de G1 vers G4 contient donc un 1, tout comme celle de G4 à G1 (le trajet est à double sens). On a alors la matrice suivante :



De même, il existe un arc de G1 vers G2 et G5 mais pas vers G3 ou G6. On peut donc compléter notre matrice :

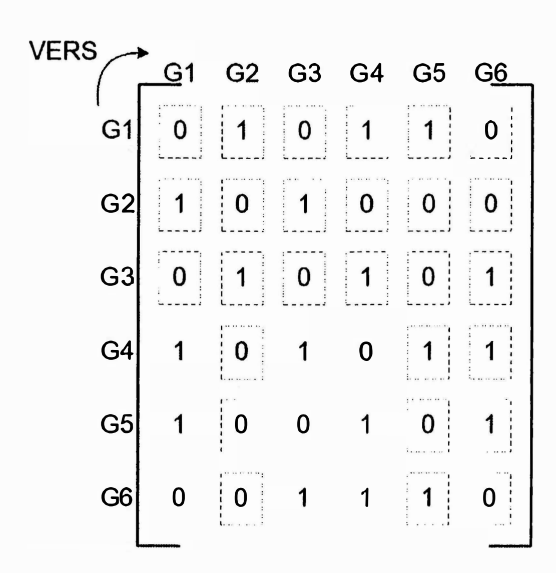


On fait de même pour tous les autres nœuds et les autres arcs :



VERS	G1	G2	G3	G4	G5	G6
G1		1	0	1	1	0
G2	1		1	0	0	0
G3	0	1		1	0	1
G4	1	0	1		1	1
G5	1	0	0	1		1
G6	0	0	1	1	1	

Il ne reste que la diagonale. Elle représente la possibilité d'aller d'un nœud à lui-même, c'est ce qu'on appelle une boucle. Ici il n'y a pas de trajet direct allant d'une gare à elle-même, on remplit donc par des 0 cette diagonale.



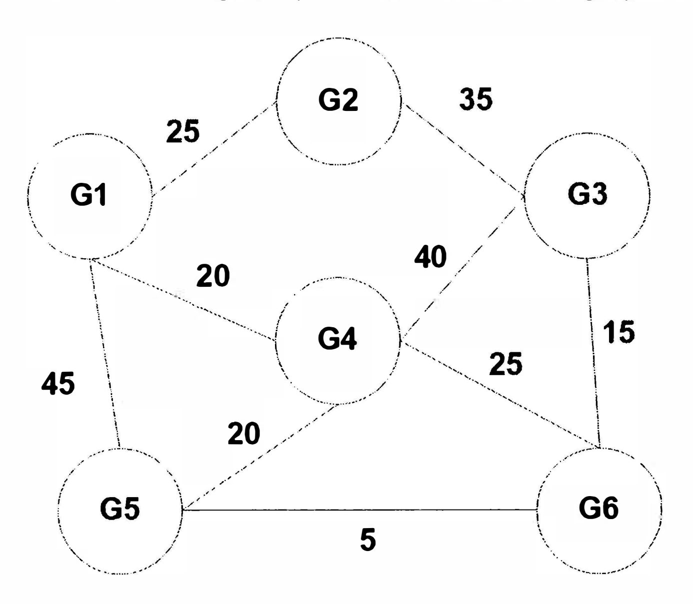
VERS	G1	G2	G3	G4	G5	G6
G1	0	1	0	1	1	0
G2	1	0	1	0	0	0
G3	0	1	0	1	0	1
G4	1	0	1	0	1	1
G5	1	0	0	1	0	1
G6	0	0	1	1	1	0

La matrice d'adjacence est alors complète.

2.3 Coût d'un chemin et matrice des longueurs

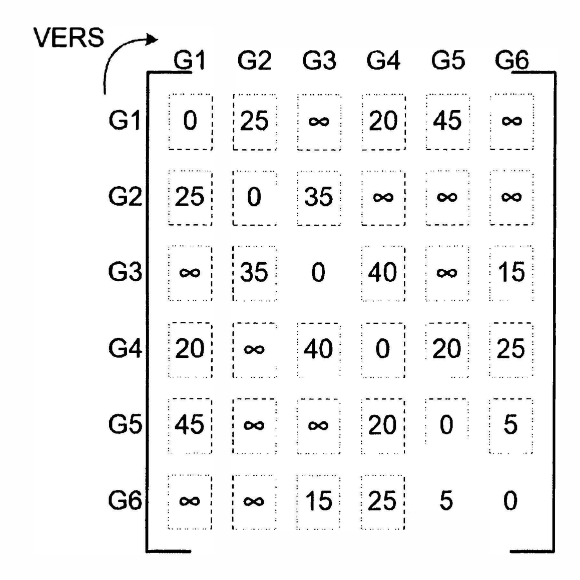
Dans le cadre de la recherche du chemin le plus court, le moins cher ou le plus rapide, on doit rajouter des informations. Celles-ci sont appelées de manière arbitraire **longueurs**, sans préciser l'unité. Il peut donc s'agir de kilomètres, d'euros, de minutes, de kilos...

Sur la représentation graphique, on rajoute les longueurs sur le dessin des arcs. Pour l'exemple sur les gares, c'est le temps nécessaire en minutes pour faire la liaison entre les gares qui est intéressant et le graphe devient :



La matrice d'adjacence devient la **matrice des longueurs**. Les "1" qui représentaient des liens sont alors remplacés par la longueur de l'arc. Quand il n'y a pas de lien, on remplace le 0 par $+\infty$, indiquant que pour aller d'un nœud à l'autre, il faudrait un temps/un kilométrage/un coût infini (et donc que ce n'est pas possible). Pour la diagonale, on met des 0, indiquant que pour aller d'un endroit à ce même endroit, on n'a pas à bouger, et donc c'est "gratuit" et immédiat.

La matrice des longueurs est donc :



C'est principalement cette matrice de longueurs qui est utilisée dans le code, même si les algorithmes seront expliqués à l'aide de représentations graphiques dans la suite.

3. Exemple en cartographie

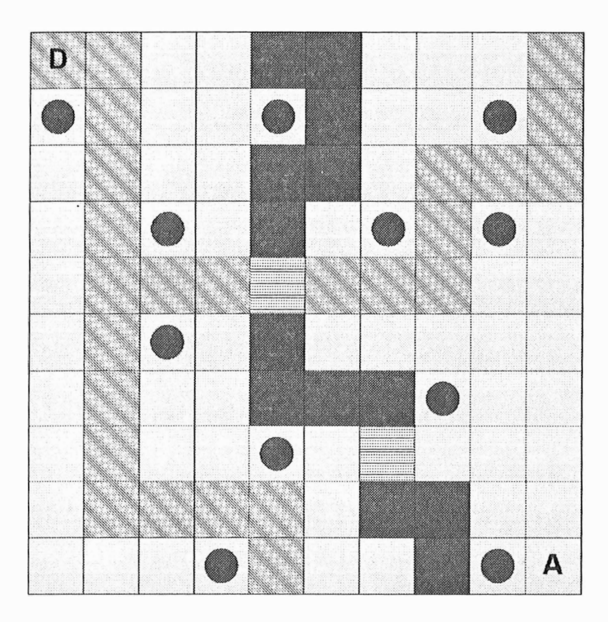
Pour étudier les différents algorithmes, nous allons nous intéresser à un petit jeu dans lequel nous contrôlons un personnage qui est un explorateur. Comme dans beaucoup de jeux de rôle, notre héros est limité à chaque tour (il n'a le droit qu'à un certain nombre de points d'action). Pour aller le plus vite d'un point à un autre, nous cherchons le chemin le plus court sur la carte, en prenant en compte les types de terrains.

Il en existe de plusieurs sortes, requérant plus ou moins d'énergie (et donc de points d'action) :

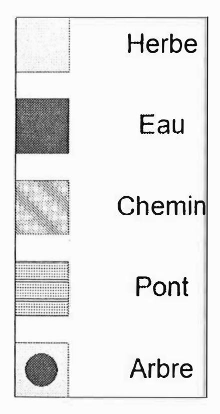
– Des chemins, qui nécessitent un point d'action par case.
– De l'herbe, nécessitant deux points d'action.

– Des ponts, nécessitant deux points d'action.
– De l'eau, infranchissable.
– Et des arbres, infranchissables aussi.

La carte est la suivante :



Et la légende :



Nous cherchons donc le chemin permettant de relier le départ (D) à l'arrivée (A), et ce en utilisant le moins de points d'action possible. Le chemin le plus court coûte 27 points d'action (en suivant le chemin et en passant le premier pont, puis en coupant par l'herbe à la fin).

4. Algorithmes naïfs de recherche de chemins

Ces premiers algorithmes ne sont pas "intelligents" : ils sont dits **naïfs**, car ils n'utilisent pas de connaissances sur le problème pour agir. Dans le pire des cas, ils testent tous les chemins possibles pour déterminer s'il existe bien un chemin entre deux nœuds.

De plus, rien n'indique que le chemin trouvé est bien le plus court. Ils sont cependant faciles à implémenter.

4.1 Parcours en profondeur

C'est l'algorithme que l'on essaie naturellement dans un labyrinthe : on cherche à avancer le plus possible, et, si on est coincé, on revient à la dernière intersection que l'on a rencontrée et on teste un nouveau chemin.

Cet algorithme ne permet pas de déterminer le chemin le plus court, mais simplement de trouver un chemin.

4.1.1 Principe et pseudo-code

Son fonctionnement est assez simple.

Tout d'abord, on choisit l'ordre des parcours des nœuds puis on applique cet ordre pour avancer un maximum. Si l'on est bloqué, on retourne en arrière, et on teste la deuxième possibilité.

Le parcours en profondeur permet donc de déterminer l'existence d'un chemin, mais il ne prend pas en compte les longueurs des arcs, et donc ne permet pas de dire si l'on a trouvé le chemin le plus court.

De plus, le résultat obtenu dépend fortement de l'ordre choisi pour le parcours du graphe, l'ordre optimal dépendant du problème et ne pouvant être déterminé a priori. Bien souvent, il n'est donc pas efficace. Dans le pire des cas, il doit même tester toutes les possibilités pour trouver un chemin.

Il est cependant facile à implémenter. En effet, nous allons conserver une liste des nœuds visités. À chaque fois que l'on se trouve sur un nœud, nous allons ajouter tous ses voisins non visités à une pile dans l'ordre choisi. Ensuite, nous dépilons le premier élément de celui-ci.

Remarque

Une pile (ou LIFO pour Last In, First Out) est une structure algorithmique dans laquelle les éléments sont ajoutés en haut, et enlevés en partant du haut, comme une pile de vêtements, de papiers... Il n'est pas possible d'enlever un élément au milieu de la pile. Ajouter un élément dessus se dit empiler, alors qu'enlever le premier se dit dépiler.

Pour faciliter la reconstruction du chemin obtenu, le prédécesseur de chaque nœud (c'est-à-dire le nœud nous ayant permis d'y aller) est conservé.

Le pseudo-code est donc le suivant :

```
// Initialisation du tableau
Créer tableau Precurseur
Pour chaque noeud n
    Precurseur[n] = null

// Création de la liste des noeuds non visités, et de la pile
Créer liste NoeudsNonvisités = ensemble des noeuds
Créer pile Avisiter
Avisiter.Empiler(départ)

// Boucle principale
Tantque AVisiter non vide
    Noeud courant = Avisiter.Depiler
    Si courant = sortie
           Fin (OK)
    Sinon
           Pour chaque voisin v de n
                  Si v dans NoeudsNonvisités
                        Enlever v de NoeudsNonvisités
                        Precurseur[v] = n
                        AVisiter.Empiler(v)
```

Remarque

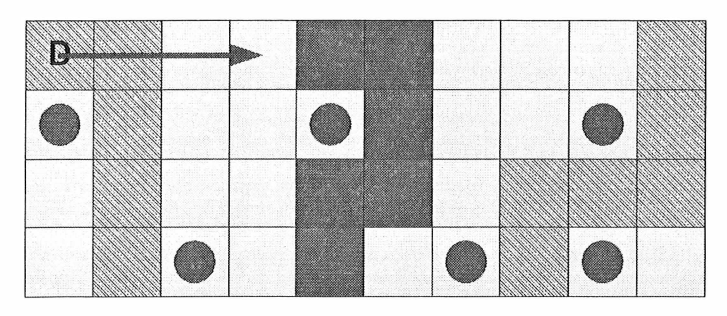
Les voisins doivent être empilés dans l'ordre inverse de l'ordre choisi, pour mettre le premier à visiter en dernier sur la pile et qu'il reste dessus.

4.1.2 Application à la carte

Pour notre carte, nous allons voir en détail comment appliquer l'ordre suivant :

- à droite,
- en bas,
- à gauche,
- et enfin en haut.

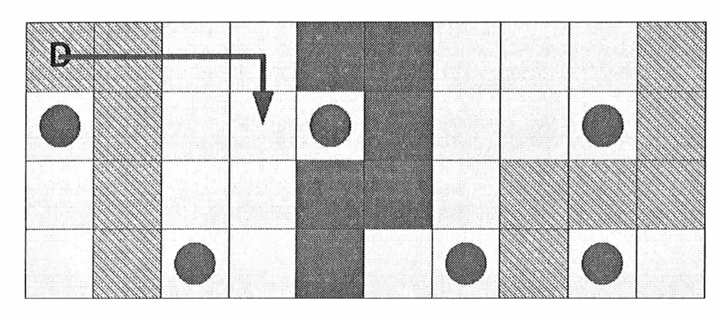
On part du nœud de départ, et on va essayer d'appliquer le plus possible le chemin allant à droite, jusqu'à être bloqué.



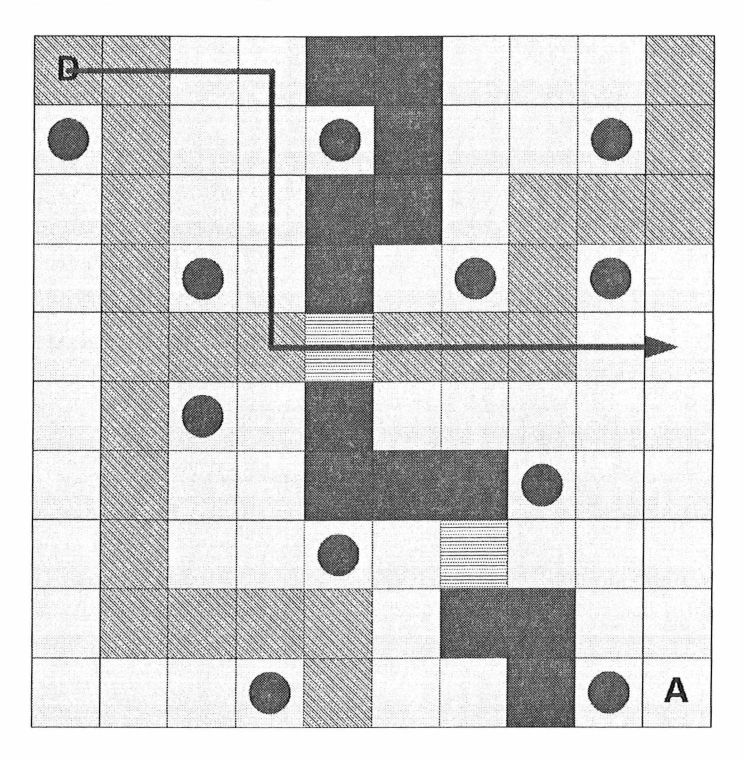
■ Remarque

Seule la partie haute de la carte est représentée pour des raisons de lisibilité.

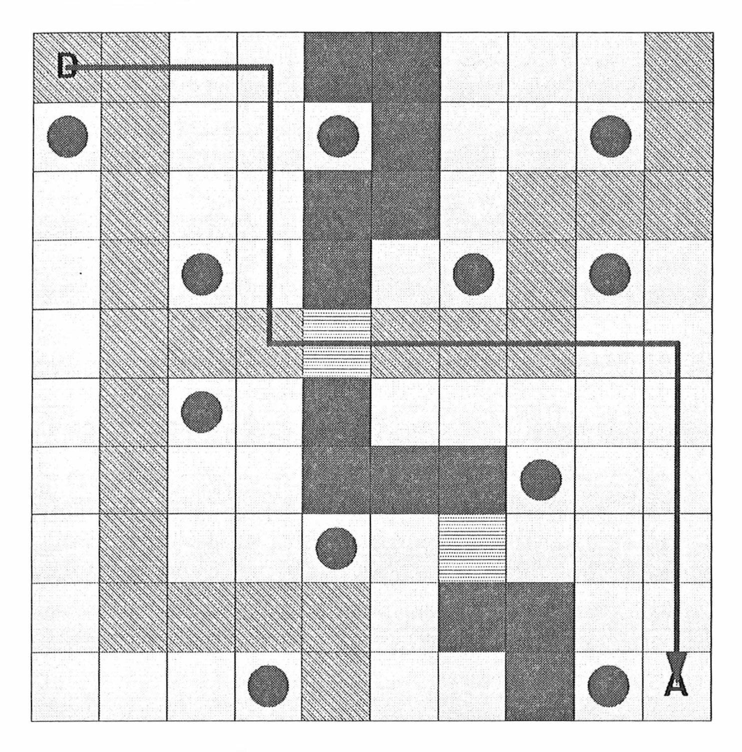
Une fois que l'on est bloqué, on va changer de direction et aller en bas (le deuxième choix) pour la case suivante.



Sur la nouvelle case, on recommence à essayer de parcourir les chemins dans notre ordre de priorité : à droite (impossible car présence d'un arbre) puis en bas. On recommence sur les cases situées dessous jusqu'à ce qu'on puisse aller à droite de nouveau.

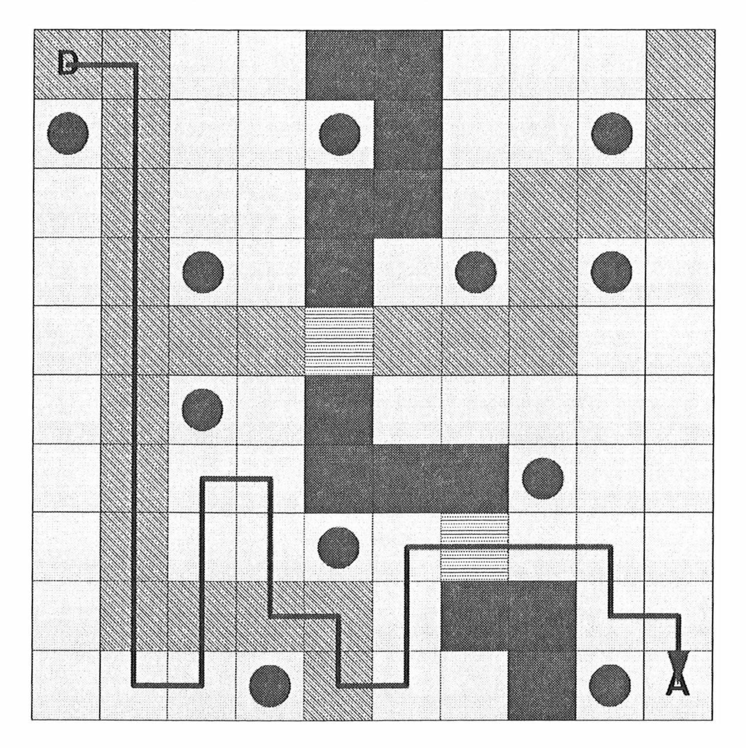


Comme on arrive à la fin de la carte et qu'il est alors impossible d'aller plus à droite, on teste la deuxième direction possible, à savoir vers le bas. On trouve alors l'arrivée.



Maintenant que l'on a trouvé un chemin, on peut calculer sa longueur. En effet, le parcours en profondeur ne prend pas en compte le poids des différents chemins. Ici, on trouve un chemin qui coûterait 32 points d'action. Il ne s'agit pas du chemin le plus court, qui ne coûte que 27 points.

De plus, on a eu "de la chance" en choisissant l'ordre de parcours des cases voisines. En effet, si on avait choisi l'ordre "bas - haut - droite - gauche", on aurait alors obtenu le chemin suivant :



Sa longueur est alors de 44, ce qui est bien supérieur au chemin trouvé précédemment.

4.2 Parcours en largeur

Le parcours en largeur est celui que la police utilise par défaut pour retrouver un coupable. On part du dernier point où celui-ci a été vu, puis on va s'en écarter progressivement en cercles concentriques.

4.2.1 Principe et pseudo-code

On part donc du nœud de départ, et on teste si l'arrivée est dans un nœud immédiatement adjacent. Si ce n'est pas le cas, alors on va tester les voisins de ces nœuds dans un ordre fixe et ainsi de suite.

Graphiquement, on va s'éloigner progressivement du point de départ, en testant toutes les cases dans un rayon donné. On parcourt donc toute la largeur de l'arbre à chaque fois, alors que dans l'algorithme précédent, on commençait par aller le plus loin possible d'abord (d'où son nom de parcours en profondeur).

Le parcours en largeur permet de tester niveau par niveau les différents nœuds accessibles. On s'éloigne donc progressivement du départ. Si tous les arcs font le même poids, on trouvera le chemin optimal, sinon rien ne permet de savoir si le chemin trouvé est le plus court.

De plus, cet algorithme n'est absolument pas efficace. En effet, il risque fort de tester de nombreux nœuds avant de trouver un chemin, car il n'utilise aucune information sur ceux-ci.

Il est cependant lui aussi facile à implémenter. Contrairement au parcours en profondeur qui utilisait une pile, celui-ci utilise une file. Le reste de l'algorithme demeure cependant le même.

Remarque

La file est une autre structure algorithmique. Elle est nommée FIFO en anglais, pour "First In, First Out" (premier arrivé, premier dehors). Dans la vie courante, on trouve des files d'attente un peu partout : le premier qui arrive fait la queue, jusqu'à pouvoir passer (à la caisse par exemple) et il sera donc le premier dehors. On dit alors qu'on enfile un élément (en l'ajoutant à la fin) et qu'on défile lorsqu'on récupère l'élément le plus au début.

Le pseudo-code est donc le suivant (les lignes qui diffèrent du parcours en profondeur sont en gras) :

```
// Initialisation du tableau
Créer tableau Precurseur
Pour chaque noeud n
    Precurseur[n] = null
```

```
// Création de la liste des noeuds non visités, et de la file
Créer liste NoeudsNonvisités = ensemble des noeuds
Créer file Avisiter
Avisiter.Enfiler(départ)

// Boucle principale
Tantque AVisiter non vide
    Noeud courant = Avisiter.Defiler
    Si courant = sortie
           Fin (OK)
    Sinon
           Pour chaque voisin v de n
                  Si v dans NoeudsNonvisités
                         Enlever v de NoeudsNonvisités
                         Precurseur[v] = n
                         AVisiter.Enfiler(v)
```

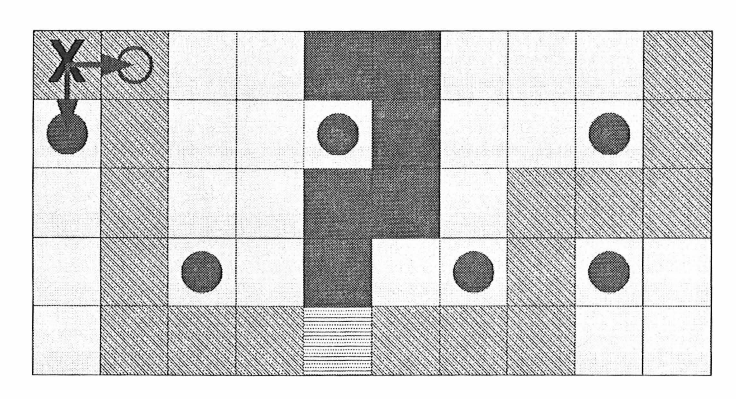
Remarque

Les voisins doivent être enfilés dans l'ordre choisi ce coup-ci.

4.2.2 Application à la carte

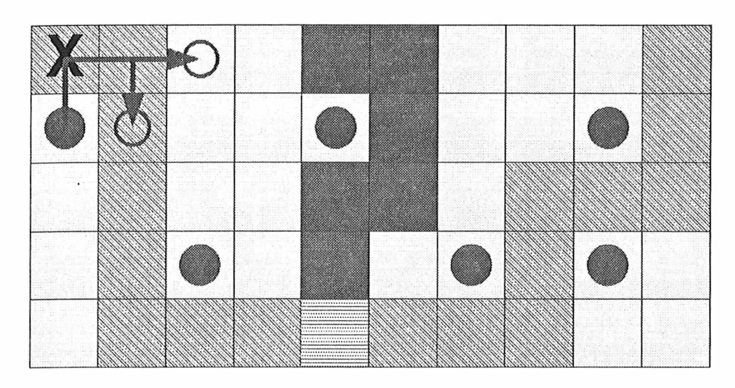
L'application à la carte est un peu plus complexe que pour le parcours en profondeur. On commence de notre point de départ, et on regarde si le point d'arrivée est autour. Seul le début de la carte est représenté pour les premières étapes.

La case notée X est notre départ. La case marquée d'un cercle est ajoutée à la file.

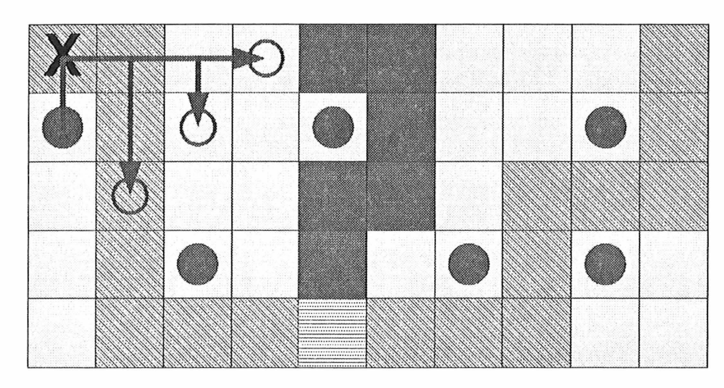


À la première étape, aucun voisin n'est le point d'arrivée. De plus, la case au sud est un arbre et il est donc impossible d'y aller. Seule la case à droite est ajoutée à la file.

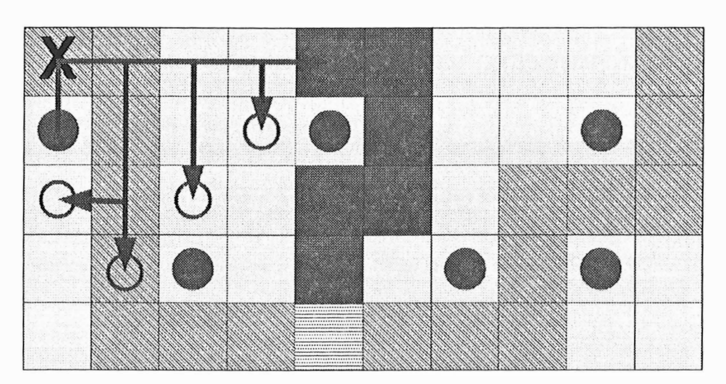
On part maintenant de cette case et on regarde ses voisines. Deux cases sont ajoutées à la file.



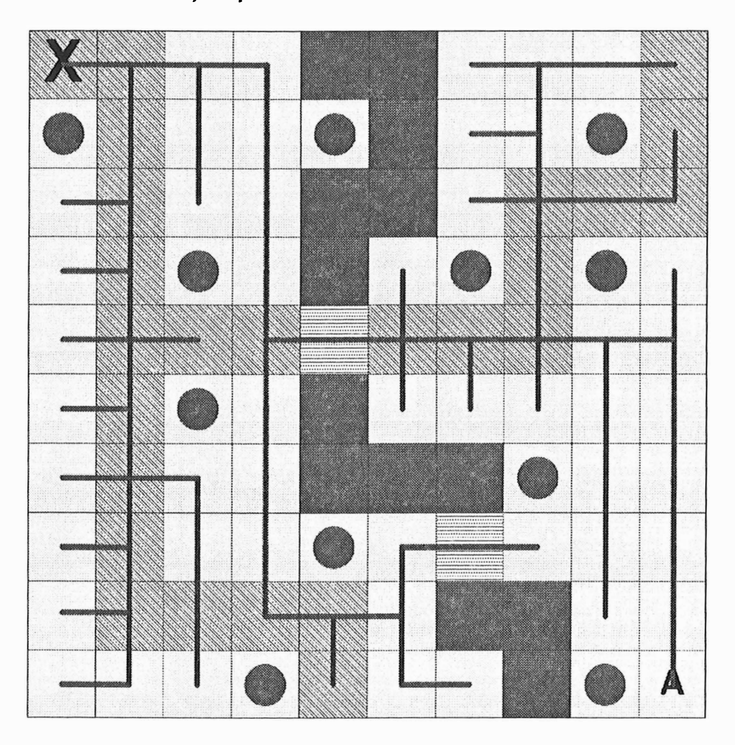
On continue, en élargissant petit à petit notre zone de recherche :



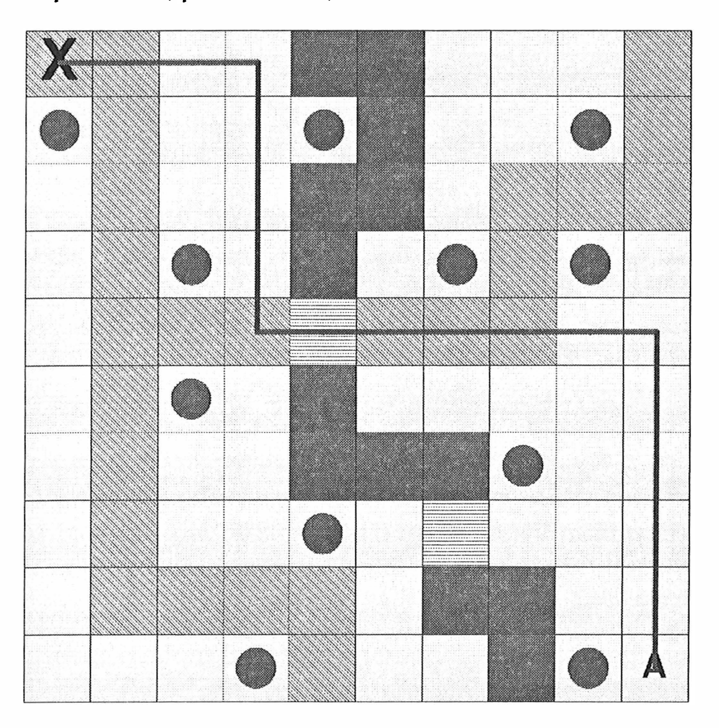
À l'itération suivante, la première case va rencontrer l'eau. Comme elle est infranchissable, on ne l'ajoute pas à la file des cases à parcourir.



On continue jusqu'à l'arrivée :



On ne conserve alors que le chemin nous ayant permis de trouver la sortie, qui est représenté dans la figure suivante. Sa taille est de 32, ce qui n'est pas l'optimum (qui est de 27).



On peut aussi voir que, lorsqu'on a enfin atteint la sortie, toutes les cases ont été explorées, ce qui a demandé beaucoup d'étapes. L'algorithme n'a donc pas du tout été efficace en termes de performances.

5. Algorithmes "intelligents"

Les parcours en profondeur et en largeur ne permettent pas de trouver le chemin le plus court, mais juste le premier qui permet de joindre le point de départ au point d'arrivée.

D'autres algorithmes existent, qui permettent de déterminer le chemin le plus court, ou au moins un chemin optimisé, et ce sans avoir forcément à tester tous les chemins possibles.

5.1 Algorithme de Bellman-Ford

L'algorithme de Bellman-Ford permet de trouver le chemin le plus court s'il existe. Il n'est pas le plus optimisé, mais c'est celui qui fonctionne dans le plus de cas. En effet, il accepte des longueurs négatives pour les arcs, et pas seulement des positives.

Remarque

De plus, s'il y a un circuit dont le poids total est négatif (et donc qui permet de diminuer le poids total), il peut le détecter. C'est important, car dans ce cas-là il n'existe pas de chemin le plus court.

5.1.1 Principe et pseudo-code

Cet algorithme va utiliser la matrice des longueurs. Son fonctionnement est itératif.

Au départ, on initialise à +∞ la longueur minimale de chaque nœud (ce qui signifie que l'on n'a pas encore trouvé de chemin jusque-là depuis l'arrivée). On va aussi garder pour chaque nœud le nœud précédent (celui qui permet d'y arriver avec la plus faible longueur) et l'initialiser au nœud vide.

On applique ensuite autant de fois que le nombre de nœuds moins 1 la même boucle. Ainsi, s'il y a 7 nœuds, on l'applique 6 fois.

À chaque itération, on suit chaque arc (u,v). On calcule la distance depuis le point de départ à v comme étant la distance du départ à u, plus la longueur de l'arc (u,v). On obtient ainsi une nouvelle longueur pour aller au nœud v que l'on compare à celle déjà enregistrée. Si cette longueur est plus faible que celle obtenue jusqu'ici, alors on change la longueur de ce nœud et on indique que son prédécesseur est maintenant u.

Remarque

Seuls les arcs partant d'un nœud dont la distance calculée est différente de +∞ ont besoin d'être utilisés. En effet, dans le cas contraire on garde une distance infinie, ce qui ne peut pas améliorer les chemins déjà trouvés.

Si à une itération il n'y a plus de changement, car aucun arc ne permet de trouver une distance plus faible que celle connue, on peut arrêter prématurément l'algorithme.

De plus, si on applique l'algorithme autant de fois que le nombre de nœuds du graphe, et qu'une modification des poids se fait encore, alors on sait qu'il existe un circuit de taille négative et qu'on ne peut donc résoudre le problème.

Le pseudo-code est donc le suivant :

```
// Initialisation des tableaux
Créer tableau Longueur_min
Créer tableau Precurseur

Pour chaque noeud n
    Longueur_min[n] = +∞
    Precurseur[n] = null
Longueur_min[depart] = 0

// Boucle principale
Pour i de 1 à nombre de noeuds - 1
    Changements = FAUX
    Pour chaque arc (u, v)
           Si Longueur_min[u] + longueur(u, v) < Longueur_min[v]
                  // On a trouvé un chemin plus court
                  Longueur_min[v] = Longueur_min[u]+longueur(u, v)
                  Precurseur[v] = u
                  Changements = VRAI
    Si Changements = FAUX
           // Aucun changement : on a fini
           Fin (OK)

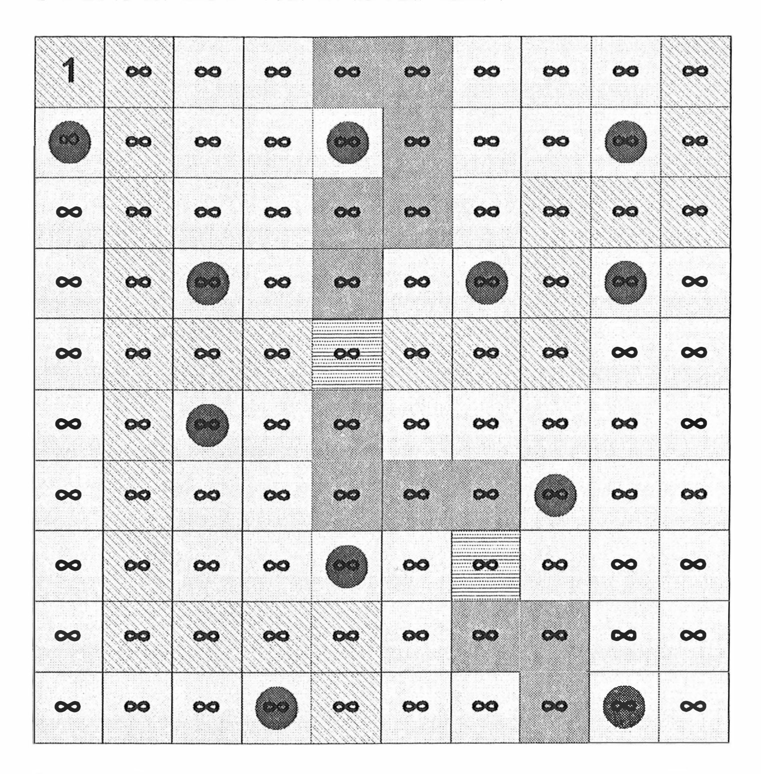
Pour chaque arc (u,v)
    Si Longueur_min[u] + longueur(u, v) < Longueur_min[v]
           // On ne peut pas résoudre ce problème
           Fin (Erreur : il y a des boucles négatives)
```

5.1.2 Application à la carte

Pour mieux comprendre l'algorithme, nous l'appliquons à notre carte avec notre explorateur qui cherche le point d'arrivée.

On va initialiser toutes les distances pour arriver à une case à la valeur +∞. Seule la case de départ possède une distance différente, égale à 1 (car c'est un chemin, dont le coût est 1).

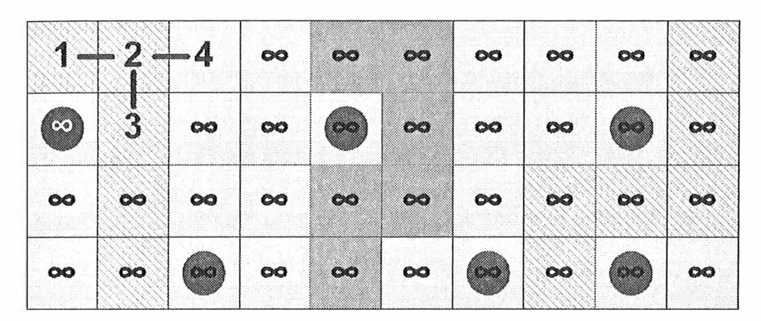
On obtient donc l'état initial suivant :



On a 100 cases, donc au maximum, on aura à faire 99 itérations pour trouver le chemin le plus court.

À la première itération, on a seulement deux arcs à appliquer : ceux sortant de la case de départ et permettant d'aller à droite et en bas. L'arc pour aller à droite vaut 1 (c'est un chemin) alors que celui pour aller au sud a une longueur de $+\infty$ (car l'arbre est infranchissable). On met donc à jour la deuxième case, pour dire que sa nouvelle distance à l'origine est 2, et que son précurseur est la première case.

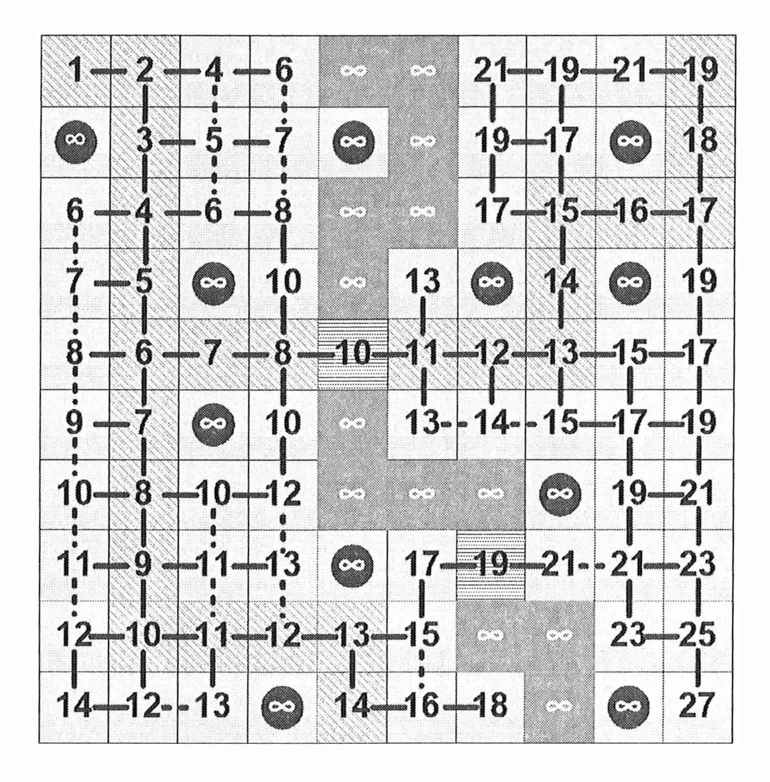
À la deuxième itération, on va maintenant appliquer les arcs sortant des deux premières cases. On trouve alors :



À l'itération suivante, on va trouver deux routes pour aller à la case au sud du 4 : une allant du 4 à cette case herbeuse, et une passant par le chemin. On conservera la route depuis le chemin qui est plus courte (5 contre 6 points d'action). Celle non conservée est marquée par des tirets.

1	2	4	6	∞	∞	∞	∞	∞	∞
∞	3	5	∞	∞	∞	∞	∞	∞	∞
∞	4	∞	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞	∞	∞

On continue pour les itérations suivantes (environ une vingtaine), jusqu'à atteindre la case d'arrivée :



On reconstruit alors un parcours nous permettant de retrouver le chemin le plus court, qui est bien de 27 points d'action.

1	2	4	6	∞	∞	21	19	21	19
∞	3	5	7	∞	∞	19	17	∞	18
6	4	6	8	∞	∞	17	15	16	17
7	5	∞	10	∞	13	∞	14	∞	19
8	6	7	8	10	11	12	13	15	17
9	7	∞	10	∞	13	14	15	17	19
10	8	10	12	∞	∞	∞	∞	19	21
11	9	11	13	∞	17	19	21	21	23
12	10	11	12	13	15	∞	∞	23	25
14	12	13	∞	14	16	18	∞	∞	27

Remarque

Plusieurs parcours nous donnent des chemins de taille 27. Nous en avons donc choisi un arbitrairement en remontant les différents prédécesseurs.

L'algorithme n'est pas efficace, car à chaque tour, tous les arcs utilisés précédemment ont été appliqués de nouveau. De plus, tous les chemins les plus courts ont été calculés, et pas seulement celui qui nous intéressait.

5.2 Algorithme de Dijkstra

L'algorithme de Dijkstra est une amélioration de l'algorithme Bellman-Ford. Il ne fonctionne que si toutes les distances sont positives, ce qui est généralement le cas dans les problèmes réels.

Il permet de choisir plus intelligemment l'ordre d'application des distances, et surtout au lieu de s'appliquer sur les arcs, il s'applique sur les nœuds, et ne revient jamais en arrière. Ainsi, chaque arc n'est appliqué qu'une seule fois.

5.2.1 Principe et pseudo-code

On commence par initialiser deux tableaux, comme pour l'algorithme de Bellman-Ford : un contenant les distances depuis le nœud initial (à $+\infty$, à l'exception du départ, à 0), et un contenant les précurseurs (tous vides).

On cherche ensuite à chaque itération le nœud qui n'a pas encore été visité et qui a la distance la plus faible au nœud de départ. On applique alors tous les arcs qui en sortent, et on modifie les distances les plus faibles si on en trouve (ainsi que le précurseur pour y arriver).

On recommence jusqu'à ce que tous les nœuds aient été utilisés, ou que l'on ait sélectionné le nœud d'arrivée.

Le pseudo-code est donc :

```
// Initialisation des tableaux
Créer tableau Longueur_min
Créer tableau Precurseur

Pour chaque noeud n
    Longueur_min[n] = +∞
    Precurseur[n] = null
Longueur_min[depart] = 0

Liste_non_visités = ensemble des noeuds

// Boucle principale
Tant que Liste_non_visités non vide
    u = noeud dans Liste_non_visités où Longueur_min[u] est min
    Pour chaque arc (u, v)
          Si Longueur_min[u] + longueur(u, v) < Longueur_min[v]
```

```
                  // On a trouvé un chemin plus court
                  Longueur_min[v] = Longueur_min[u]+longueur(u, v)
                  Precurseur[v] = u
    Enlever u de Liste_non_visités
    Si u = sortie
        Fin (OK)
```

La grosse différence par rapport à Bellman-Ford est donc l'application des distances nœud par nœud et une seule fois, ce qui apporte une grosse optimisation en termes de temps et de nombre de calculs.

5.2.2 Application à la carte

On applique maintenant notre algorithme à notre carte en 2D. Les cases déjà visitées sont grisées. La case en cours est soulignée. L'initialisation est la même que pour Bellman-Ford :

1	∞	∞	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞	∞	∞

La première case est sélectionnée car c'est la seule qui contient une distance non nulle. On applique donc ce nœud-là, et on trouve la distance à la case à droite (la case en bas contient un arbre, il est donc impossible d'y aller).

1	2	∞	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞	∞	∞

La première case est maintenant visitée, c'est donc la deuxième qui sert de départ à l'itération suivante :

1	2	4	∞	∞	∞	∞	∞	∞	∞
∞	3	∞	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞	∞	∞

La case ayant la plus faible distance à l'origine est celle contenant 3 sur le chemin. C'est de cette case que l'on va maintenant partir.

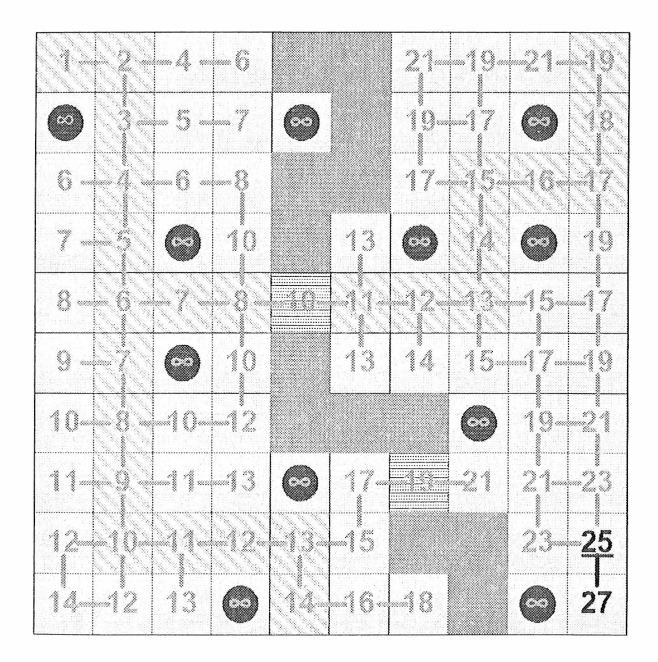
1	2	4	∞	∞	∞	∞	∞	∞	∞
∞	3	5	∞	∞	∞	∞	∞	∞	∞
∞	4	∞	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞	∞	∞

Ce coup-ci, deux cases ont une distance de 4. On applique donc les arcs sortants dans l'ordre de lecture (donc de gauche à droite puis de haut en bas). On commence donc par appliquer celui qui est au nord. Comme en passant par le nord, on arrive sur la case actuellement marquée 5 avec 6 points d'actions, l'arc n'est pas pris en compte.

On a donc la situation suivante :

1	2	4	6	∞	∞	∞	∞	∞	∞
∞	3	5	∞	∞	∞	∞	∞	∞	∞
∞	4	∞	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞	∞	∞

On continue au fur et à mesure des itérations, jusqu'à trouver la case d'arrivée. On a alors la situation suivante :



On obtient exactement les mêmes résultats qu'avec Bellman-Ford, mais en ayant effectué beaucoup moins de calculs, car chaque arc n'a été appliqué (et donc les distances calculées) qu'une seule fois. Les chemins trouvés sont les mêmes.

5.3 Algorithme A*

L'algorithme A* (que l'on prononce "A étoile", ou à l'anglaise "A star") fonctionne sur les graphes dans lesquels toutes les distances sont positives (comme pour Dijkstra).

Contrairement aux deux algorithmes précédents, A* ne teste pas tous les chemins, et ne peut pas assurer que le chemin trouvé est le plus court. Il s'agit simplement d'une approximation, qui donne généralement le meilleur chemin ou un des meilleurs. Il existe cependant des cas, comme les labyrinthes, pour lesquels A* est très peu voire pas du tout efficace. Dans des espaces ouverts avec peu d'obstacles comme notre carte, il est par contre très performant.

5.3.1 Principe et pseudo-code

L'algorithme A* utilise des informations sur l'endroit où se trouve le but pour choisir la direction à suivre. En effet, il se base sur le fait que la distance la plus courte est généralement la ligne droite, et non en faisant de nombreux détours.

C'est l'algorithme que l'on utilise intuitivement quand on est perdu dans une ville : on cherche dans quelle direction on doit aller, et on essaie de suivre cette direction. Si ce n'est pas possible (par exemple parce qu'il n'y a pas de route), alors on va légèrement s'en écarter puis revenir vers notre but.

Il nous faut donc une façon d'estimer la distance restante entre chaque nœud et la sortie. Plus cette approximation sera précise et plus les résultats le seront aussi. Par contre, celle-ci ne doit jamais surestimer la distance, mais la sous-estimer ou donner la distance exacte. Rien ne dit dans l'algorithme comment choisir cette distance. Celle-ci dépend du problème.

Comme pour les algorithmes précédents, on garde la longueur minimale pour arriver à un nœud et son prédécesseur. De plus, la liste des nœuds non encore visités est stockée, et pour chaque nœud, la distance estimée à l'arrivée.

Ensuite, parmi les nœuds qui ne sont pas déjà visités, on cherche celui qui a la distance totale la plus faible. Celle-ci correspond à la somme de la distance depuis l'origine avec la distance estimée à la sortie. On a alors le nœud le plus prometteur. On applique donc les arcs qui en sortent et on le note comme visité, puis on continue.

```
// Initialisation des tableaux
Créer tableau Longueur_min
Créer tableau Precurseur
Créer tableau DistanceEstimee

Pour chaque noeud n
    Longueur_min[n] = +∞
    Precurseur[n] = null
    DistanceEstimee[n] = heuristique à calculer
Longueur_min[depart] = 0

Liste_non_visités = ensemble des noeuds

// Boucle principale
Tant que Liste_non_visités non vide
    u = noeud dans Liste_non_visités où Longueur_min[u]
+DistanceEstimee[u] est min
    Enlever u de Liste_non_visités
    Pour chaque arc (u, v)
           Si Longueur_min[u] + longueur(u, v) < Longueur_min[v]
                 // On a trouvé un chemin plus court
                 Longueur_min[v] = Longueur_min[u]+longueur(u, v)
                 Precurseur[v] = u

    Si u = sortie
           Fin (OK)
```

Dans le pire des cas, l'algorithme A* va être équivalent à l'algorithme de Dijkstra.

5.3.2 Application à la carte

Pour notre carte, la distance d'approximation choisie est la distance de Manhattan. Elle est définie par le nombre de cases horizontales ajouté au nombre de cases verticales qui nous séparent de l'arrivée.

Il s'agit forcément d'une sous-estimation. En effet, cela suppose que toutes les cases ont un coût de 1 point d'action, or les ponts et les prairies ont un coût de deux, et certaines cases sont infranchissables (et ont donc une distance infinie). La distance choisie est donc adaptée car elle sous-estime la distance réelle.

Voici les distances estimées pour chaque case :

18	17	16	15	14	13	12	11	10	9
17	16	15	14	13	12	11	10	9	8
16	15	14	13	12	11	10	9	8	7
15	14	13	12	11	10	9	8	7	6
14	13	12	11	10	9	8	7	6	5
13	12	11	10	9	8	7	6	5	4
12	11	10	9	8	7	6	5	4	3
11	10	9	8	7	6	5	4	3	2
10	9	8	7	6	5	4	3	2	1
9	8	7	6	5	4	3	2	1	A

À l'initialisation, aucun nœud n'a été visité, et le seul possédant une distance qui ne vaut pas $+\infty$ est la case de départ (qui est à 1).

À partir du départ, on calcule la distance réelle aux deux cases autour (en bas et à droite). Celle dessous contient un arbre et n'est donc pas atteignable, sa distance reste donc à $+\infty$. Celle qui se trouve à droite est un chemin, son coût est donc de 1. La distance au départ est donc de 2.

On obtient les distances suivantes (les flèches permettent d'indiquer les prédécesseurs, et les distances estimées restantes sont entre parenthèses) :

1 (18)	→ 2 (17)	∞ (16)	∞ (15)	∞ (14)	∞ (13)	∞ (12)	∞ (11)	∞ (10)	∞ (9)
∞ (17)	∞ (16)	∞ (15)	∞ (14)	∞ (13)	∞ (12)	∞ (11)	∞ (10)	∞ (9)	∞ (8)
∞ (16)	∞ (15)	∞ (14)	∞ (13)	∞ (12)	∞ (11)	∞ (10)	∞ (9)	∞ (8)	∞ (7)
∞ (15)	∞ (14)	∞ (13)	∞ (12)	∞ (11)	∞ (10)	∞ (9)	∞ (8)	∞ (7)	∞ (6)
∞ (14)	∞ (13)	∞ (12)	∞ (11)	∞ (10)	∞ (9)	∞ (8)	∞ (7)	∞ (6)	∞ (5)
∞ (13)	∞ (12)	∞ (11)	∞ (10)	∞ (9)	∞ (8)	∞ (7)	∞ (6)	∞ (5)	∞ (4)
∞ (12)	∞ (11)	∞ (10)	∞ (9)	∞ (8)	∞ (7)	∞ (6)	∞ (5)	∞ (4)	∞ (3)
∞ (11)	∞ (10)	∞ (9)	∞ (8)	∞ (7)	∞ (6)	∞ (5)	∞ (4)	∞ (3)	∞ (2)
∞ (10)	∞ (9)	∞ (8)	∞ (7)	∞ (6)	∞ (5)	∞ (4)	∞ (3)	∞ (2)	∞ (1)
∞ (9)	∞ (8)	∞ (7)	∞ (6)	∞ (5)	∞ (4)	∞ (3)	∞ (2)	∞ (1)	∞ (0)

Le nœud qui a la distance totale la plus faible et non encore visité est le deuxième nœud (distance totale de 19). Il possède deux voisins, et on calcule donc les distances pour y arriver, à savoir 2 pour la case en herbe et 1 pour la case en chemin. On obtient les nouvelles distances suivantes (seul le haut de la carte est présent) :

1 (18)	2 (17)	4 (16)	∞ (15)	∞ (14)	∞ (13)	∞ (12)	∞ (11)	∞ (10)	∞ (9)
∞ (17)	3 (16)	∞ (15)	∞ (14)	∞ (13)	∞ (12)	∞ (11)	∞ (10)	∞ (9)	∞ (8)
∞ (16)	∞ (15)	∞ (14)	∞ (13)	∞ (12)	∞ (11)	∞ (10)	∞ (9)	∞ (8)	∞ (7)
∞ (15)	∞ (14)	∞ (13)	∞ (12)	∞ (11)	∞ (10)	∞ (9)	∞ (8)	∞ (7)	∞ (6)

Si on ajoute les distances depuis l'origine aux distances estimées pour arriver à la sortie, on voit que c'est le nœud sur le chemin qui semble le plus prometteur. En effet, on obtient une distance totale de 3+ 16 = 19 contre 4+ 16 = 20. C'est celui-là que l'on va utiliser donc à la prochaine itération.

1 (18)	2 (17)	4 (16)	∞ (15)	∞ (14)	∞ (13)	∞ (12)	∞ (11)	∞ (10)	∞ (9)
∞ (17)	3 (16)	5 (15)	∞ (14)	∞ (13)	∞ (12)	∞ (11)	∞ (10)	∞ (9)	∞ (8)
∞ (16)	4 (15)	∞ (14)	∞ (13)	∞ (12)	∞ (11)	∞ (10)	∞ (9)	∞ (8)	∞ (7)
∞ (15)	∞ (14)	∞ (13)	∞ (12)	∞ (11)	∞ (10)	∞ (9)	∞ (8)	∞ (7)	∞ (6)

De nouveau, c'est le nœud sur le chemin qui est le plus prometteur, avec une distance totale de 4+15 = 19. On utilise donc ce nœud-là et on applique les arcs en sortant.

1 (18)	2 (17)	4 (16)	∞ (15)	∞ (14)	∞ (13)	∞ (12)	∞ (11)	∞ (10)	∞ (9)
∞ (17)	3 (16)	5 (15)	∞ (14)	∞ (13)	∞ (12)	∞ (11)	∞ (10)	∞ (9)	∞ (8)
6 (16)	4 (15)	6 (14)	∞ (13)	∞ (12)	∞ (11)	∞ (10)	∞ (9)	∞ (8)	∞ (7)
∞ (15)	5 (14)	∞ (13)	∞ (12)	∞ (11)	∞ (10)	∞ (9)	∞ (8)	∞ (7)	∞ (6)

On continue sur deux nouvelles itérations :

1 (18)	2 (17)	4 (16)	∞ (15)	∞ (14)	∞ (13)	∞ (12)	∞ (11)	∞ (10)	∞ (9)
∞ (17)	3 (16)	5 (15)	∞ (14)	∞ (13)	∞ (12)	∞ (11)	∞ (10)	∞ (9)	∞ (8)
6 (16)	4 (15)	6 (14)	∞ (13)	∞ (12)	∞ (11)	∞ (10)	∞ (9)	∞ (8)	∞ (7)
7 (15)	5 (14)	∞ (13)	∞ (12)	∞ (11)	∞ (10)	∞ (9)	∞ (8)	∞ (7)	∞ (6)
8 (14)	6 (13)	7 (12)	∞ (11)	∞ (10)	∞ (9)	∞ (8)	∞ (7)	∞ (6)	∞ (5)
∞ (13)	7 (12)	∞ (11)	∞ (10)	∞ (9)	∞ (8)	∞ (7)	∞ (6)	∞ (5)	∞ (4)

Ce coup-ci, on a deux nœuds à égalité : les deux nœuds à 7 sur les chemins ont une distance estimée à la sortie de 12, soit une distance totale de 19. Comme on ne peut pas les départager, l'algorithme va utiliser le premier nœud dans l'ordre de lecture, à savoir celui qui est à droite.

On applique de nouveau l'algorithme pendant deux nouvelles itérations :

1 (18)	2 (17)	4 (16)	∞ (15)	∞ (14)	∞ (13)	∞ (12)	∞ (11)	∞ (10)	∞ (9)
∞ (17)	3 (16)	5 (15)	∞ (14)	∞ (13)	∞ (12)	∞ (11)	∞ (10)	∞ (9)	∞ (8)
6 (16)	4 (15)	6 (14)	∞ (13)	∞ (12)	∞ (11)	∞ (10)	∞ (9)	∞ (8)	∞ (7)
7 (15)	5 (14)	∞ (13)	10 (12)	∞ (11)	∞ (10)	∞ (9)	∞ (8)	∞ (7)	∞ (6)
8 (14)	6 (13)	7 (12)	8 (11)	10 (10)	∞ (9)	∞ (8)	∞ (7)	∞ (6)	∞ (5)
∞ (13)	7 (12)	∞ (11)	10 (10)	∞ (9)	∞ (8)	∞ (7)	∞ (6)	∞ (5)	∞ (4)
∞ (12)	∞ (11)	∞ (10)	∞ (9)	∞ (8)	∞ (7)	∞ (6)	∞ (5)	∞ (4)	∞ (3)
∞ (11)	∞ (10)	∞ (9)	∞ (8)	∞ (7)	∞ (6)	∞ (5)	∞ (4)	∞ (3)	∞ (2)
∞ (10)	∞ (9)	∞ (8)	∞ (7)	∞ (6)	∞ (5)	∞ (4)	∞ (3)	∞ (2)	∞ (1)
∞ (9)	∞ (8)	∞ (7)	∞ (6)	∞ (5)	∞ (4)	∞ (3)	∞ (2)	∞ (1)	∞ (0)

Ce coup-ci, on a rencontré un pont, qui a un coût de 2. Avec une distance estimée à la sortie de 10, la distance totale est de 20, ce qui est supérieur au chemin laissé de côté en bas. On repart donc de celui-là, qui a une distance totale de 7+12 = 19 points d'action. Et on continue tant qu'il existe des cases avec un total de 19, jusqu'à obtenir le résultat suivant :

1 (18)	2 (17)	4 (16)	∞ (15)	∞ (14)	∞ (13)	∞ (12)	∞ (11)	∞ (10)	∞ (9)
∞ (17)	3 (16)	5 (15)	∞ (14)	∞ (13)	∞ (12)	∞ (11)	∞ (10)	∞ (9)	∞ (8)
6 (16)	4 (15)	6 (14)	∞ (13)	∞ (12)	∞ (11)	∞ (10)	∞ (9)	∞ (8)	∞ (7)
7 (15)	5 (14)	∞ (13)	10 (12)	∞ (11)	∞ (10)	∞ (9)	∞ (8)	∞ (7)	∞ (6)
8 (14)	6 (13)	7 (12)	8 (11)	10 (10)	∞ (9)	∞ (8)	∞ (7)	∞ (6)	∞ (5)
9 (13)	7 (12)	∞ (11)	10 (10)	∞ (9)	∞ (8)	∞ (7)	∞ (6)	∞ (5)	∞ (4)
10 (12)	8 (11)	10 (10)	∞ (9)	∞ (8)	∞ (7)	∞ (6)	∞ (5)	∞ (4)	∞ (3)
11 (11)	9 (10)	11 (9)	14 (8)	∞ (7)	∞ (6)	∞ (5)	∞ (4)	∞ (3)	∞ (2)
12 (10)	10 (9)	11 (8)	12 (7)	13 (6)	15 (5)	∞ (4)	∞ (3)	∞ (2)	∞ (1)
∞ (9)	12 (8)	13 (7)	∞ (6)	14 (5)	16 (4)	∞ (3)	∞ (2)	∞ (1)	∞ (0)

Comme il n'y a plus de chemins à 19, on repart avec les chemins dont la somme est égale à 20, donc du nœud en haut (4+16). Et à chaque itération, on va avancer à partir de la première case qui a une distance de 20.

1 (18)	2 (17)	4 (16)	6 (15)	∞ (14)	∞ (13)	∞ (12)	∞ (11)	∞ (10)	∞ (9)
∞ (17)	3 (16)	5 (15)	7 (14)	∞ (13)	∞ (12)	∞ (11)	∞ (10)	∞ (9)	∞ (8)
6 (16)	4 (15)	6 (14)	8 (13)	∞ (12)	∞ (11)	∞ (10)	∞ (9)	∞ (8)	∞ (7)
7 (15)	5 (14)	∞ (13)	10 (12)	∞ (11)	13 (10)	∞ (9)	14 (8)	∞ (7)	∞ (6)
8 (14)	6 (13)	7 (12)	8 (11)	10 (10)	11 (9)	12 (8)	13 (7)	15 (6)	∞ (5)
9 (13)	7 (12)	∞ (11)	10 (10)	∞ (9)	13 (8)	14 (7)	15 (6)	∞ (5)	∞ (4)
10 (12)	8 (11)	10 (10)	12 (9)	∞ (8)	∞ (7)	∞ (6)	∞ (5)	∞ (4)	∞ (3)
11 (11)	9 (10)	11 (9)	13 (8)	∞ (7)	17 (6)	∞ (5)	∞ (4)	∞ (3)	∞ (2)
12 (10)	10 (9)	11 (8)	12 (7)	13 (6)	15 (5)	∞ (4)	∞ (3)	∞ (2)	∞ (1)
∞ (9)	12 (8)	13 (7)	∞ (6)	14 (5)	16 (4)	18 (3)	∞ (2)	∞ (1)	∞ (0)

Il n'y a plus de chemins de taille 20, on passe donc aux chemins de taille totale 21, puis 22, 23... Lorsque l'on arrive au but, on a donc le résultat suivant :

1 (18)	2 (17)	4 (16)	6 (15)	∞ (14)	∞ (13)	∞ (12)	∞ (11)	∞ (10)	19 (9)
∞ (17)	3 (16)	5 (15)	7 (14)	∞ (13)	∞ (12)	∞ (11)	17 (10)	∞ (9)	18 (8)
6 (16)	4 (15)	6 (14)	8 (13)	∞ (12)	∞ (11)	17 (10)	15 (9)	16 (8)	17 (7)
7 (15)	5 (14)	∞ (13)	10 (12)	∞ (11)	13 (10)	∞ (9)	14 (8)	∞ (7)	19 (6)
8 (14)	6 (13)	7 (12)	8 (11)	10 (10)	11 (9)	12 (8)	13 (7)	15 (6)	17 (5)
9 (13)	7 (12)	∞ (11)	10 (10)	∞ (9)	13 (8)	14 (7)	15 (6)	17 (5)	19 (4)
10 (12)	8 (11)	10 (10)	12 (9)	∞ (8)	∞ (7)	∞ (6)	∞ (5)	19 (4)	21 (3)
11 (11)	9 (10)	11 (9)	13 (8)	∞ (7)	17 (6)	19 (5)	21 (4)	21 (3)	23 (2)
12 (10)	10 (9)	11 (8)	12 (7)	13 (6)	15 (5)	∞ (4)	∞ (3)	23 (2)	25 (1)
14 (9)	12 (8)	13 (7)	∞ (6)	14 (5)	16 (4)	18 (3)	∞ (2)	∞ (1)	27 (0)

On a atteint la sortie : l'algorithme s'arrête donc. On trouve une distance de 27, ce qui est bien l'optimum (et donc le même parcours que pour les deux algorithmes précédents), sauf que ce coup-ci, toutes les cases n'ont pas été évaluées (celles en haut à droite ont encore une distance infinie à la sortie car non calculée).

Plus l'espace est grand autour du point de départ et plus l'optimisation se fait sentir. Par contre, en cas de chemins très complexes avec de fortes contraintes comme dans un labyrinthe, l'algorithme n'est pas vraiment performant.

6. Implémentations

Nous allons passer à l'implémentation de ces algorithmes. Le code sera cependant très générique, ce qui permettra facilement d'ajouter des nouvelles méthodes de résolution ou de nouveaux problèmes à résoudre.

Nous l'appliquerons au problème de la carte ensuite, à travers une application console.

6.1 Nœuds, arcs et graphes

La première étape est la définition de nos graphes. Nous allons commencer par les nœuds, puis nous verrons les arcs qui les relient et enfin le graphe complet.

6.1.1 Implémentation des nœuds

Les nœuds sont les structures de base de nos graphes. Cependant, le contenu réel d'un nœud dépend fortement du problème à résoudre : il peut s'agir de gares, de cases sur une grille, de serveurs, de villes...

Nous créons donc une classe abstraite **Node**, qui contiendra les informations nécessaires pour les algorithmes. Cette classe doit être héritée pour la résolution pratique d'un problème.

Les nœuds ont besoin de trois informations :

- Le précurseur, qui est aussi un nœud.
- La distance depuis le départ.
- La distance estimée à la sortie (si nécessaire).

Nous utilisons des propriétés. Pour les deux premières, elles possèdent des valeurs par défaut.

```
public abstract class Node
{
    private Node precursor = null;
    internal Node Precursor
    {
           get
           {
```

```
                return precursor;
            }
            set
            {
                precursor = value;
        }
    }

    private double distanceFromBegin = double.PositiveInfinity;
    internal double DistanceFromBegin
    {
            get
            {
                return distanceFromBegin;
            }
            set
            {
                distanceFromBegin = value;
            }
    }

    internal double EstimatedDistance { get; set; }
}
```

6.1.2 Classe représentant les arcs

Une fois les nœuds définis, nous pouvons définir les arcs, grâce à la classe **Arc**. Ceux-ci contiennent trois propriétés :

– Le nœud de départ de l'arc.

– Le nœud d'arrivée.

– La longueur ou coût de l'arc.

Un constructeur est ajouté pour initialiser plus rapidement les trois propriétés :

```
public class Arc
{
    internal Node FromNode { get; set; }
    internal Node ToNode { get; set; }
    internal double Cost { get; set; }

    internal Arc(Node _fromNode, Node _toNode, double _costNode)
```

```
    {
        FromNode = _fromNode;
        ToNode = _toNode;
        Cost = _costNode;
    }
}
```

6.1.3 Interface des graphes

Nous passons maintenant aux graphes. Nous utilisons une interface, **Graph**, qui contiendra toutes les méthodes que devra définir chaque graphe. En effet, la plupart de ces méthodes seront fortement dépendantes du problème.

```
using System;
using System.Collections.Generic;

public interface Graph
{
    // Code ici
}
```

Nous avons tout d'abord besoin de deux méthodes pour obtenir le nœud de départ ou d'arrivée :

```
    Node BeginningNode();

    Node ExitNode();
```

Nous ajoutons aussi deux méthodes pour récupérer tous les nœuds sous forme de liste et deux pour récupérer tous les arcs. La première ne prend pas de paramètre et renvoie donc la liste complète alors que la deuxième prend un nœud et ne renvoie que les nœuds adjacents ou les arcs en sortant.

```
    List<Node> NodesList();

    List<Node> NodesList(Node _currentNode);

    List<Arc> ArcsList();

    List<Arc> ArcsList(Node _currentNode);
```

Quelques fonctions utilitaires sont ajoutées pour :

- Compter le nombre de nœuds.
- Retourner la distance entre deux nœuds.
- Calculer la distance estimée à la sortie.
- Reconstruire le chemin à partir des prédécesseurs.
- Remettre le graphe dans son état initial.

```
int NodesCount();

double CostBetweenNodes(Node _fromNode, Node _toNode);

String ReconstructPath();

void ComputeEstimatedDistanceToExit();

void Clear();
```

Nos graphes étant maintenant codés, nous allons pouvoir passer au reste.

6.2 Fin du programme générique

Pour terminer le programme générique, il nous manque deux éléments : une interface pour l'IHM et une classe abstraite dont hériteront les différents algorithmes.

6.2.1 IHM

Le programme est ici utilisé dans une console, mais il pourrait être utilisé dans une interface graphique, ou être envoyé à travers un réseau sous la forme d'un service web. On doit donc séparer le programme générique des sorties.

Nous créons donc une interface **IHM** pour la définir. Celle-ci a juste à afficher le résultat fourni sous la forme d'un chemin et de la distance obtenue pour le chemin.

```
public interface IHM
{
    void PrintResult(string _path, double _distance);
}
```

6.2.2 Algorithme générique

La dernière classe est celle de l'algorithme générique. Il s'agit de la classe abstraite **Algorithm**, dont hérite chacun des cinq algorithmes que nous avons vus.

Cette classe contient tout d'abord deux propriétés pour le graphe à traiter et l'IHM pour la sortie, ainsi qu'un constructeur pour les initialiser :

```
public abstract class Algorithm
{
    protected Graph graph;
    protected IHM ihm;

    public Algorithm(Graph _graph, IHM _ihm)
    {
           graph = _graph;
           ihm = _ihm;
    }
}
```

Il nous faut ensuite la méthode principale, `Solve()`.

Remarque

Celle-ci suit le design pattern "patron de méthode", c'est-à-dire qu'elle nous permet de fixer le comportement général de la méthode. Les descendants n'auront qu'à redéfinir certaines méthodes.

Cette méthode n'a que trois étapes :

- Réinitialiser le problème et donc le graphe (au cas où).
- Lancer l'algorithme.
- Afficher le résultat via l'IHM.

La méthode `Run` est abstraite, c'est elle et uniquement elle qui est à redéfinir.

```
    public void Solve()
    {
           // Nettoyage
           graph.Clear();

           // Lancement de l'algorithme
           Run();
```

```
            // Affichage du résultat
            ihm.PrintResult(graph.ReconstructPath(),
graph.ExitNode().DistanceFromBegin);
    }

    protected abstract void Run();
```

6.3 Codage des différents algorithmes

Le programme générique est terminé. Il ne reste plus qu'à coder les différents algorithmes, qui héritent tous de la classe `Algorithm`. Il est ainsi possible d'ajouter des nouveaux algorithmes facilement.

De plus, pour des raisons de lisibilité du code, les implémentations sont très proches des pseudo-codes. Il est possible de les optimiser encore si besoin est.

6.3.1 Recherche en profondeur

Nous commençons par la recherche de chemins en profondeur. Comme pour chaque algorithme, cette classe **`DepthFirst`** hérite de la classe abstraite `Algorithm`. Le constructeur se contente d'appeler celui de la classe mère.

Au niveau de la méthode `Run`, qui est le cœur de notre classe, le pseudo-code vu précédemment est reproduit. Celui-ci consiste à conserver la liste de tous les nœuds non encore visités, et une pile de nœuds. On part du nœud de départ, et on ajoute à la pile tous les voisins en les mettant à jour (précurseur et distance depuis l'origine).

Si le nœud que l'on dépile est la sortie, alors on a fini.

On utilise la classe `Stack` pour gérer la pile, qui a deux méthodes importantes pour nous : `Push` qui ajoute un élément à la liste et `Pop` qui récupère le premier élément en haut de celle-ci.

Le code est donc le suivant :

```
using System.Collections.Generic;

public class DepthFirst : Algorithm
{
    public DepthFirst(Graph _graph, IHM _ihm) : base(_graph, _ihm) {}

    protected override void Run()
    {
        // Création de la liste des noeuds non visités, et de la pile
        List<Node> notVisitedNodes = graph.NodesList();
        Stack<Node> nodesToVisit = new Stack<Node>();
        nodesToVisit.Push(graph.BeginningNode());
        notVisitedNodes.Remove(graph.BeginningNode());

        Node exitNode = graph.ExitNode();

        bool exitReached = false;
        // Boucle principale
        while (nodesToVisit.Count != 0 && !exitReached)
        {
            Node currentNode = nodesToVisit.Pop();
            if (currentNode.Equals(exitNode))
            {
                // On a fini
                exitReached = true;
            }
            else
            {
                // On ajoute les voisins
                foreach (Node node in
graph.NodesList(currentNode))
                {
                    if (notVisitedNodes.Contains(node))
                    {
    notVisitedNodes.Remove(node);
                        node.Precursor = currentNode;
                        node.DistanceFromBegin =
currentNode.DistanceFromBegin + graph.CostBetweenNodes(currentNode,
node);
                        nodesToVisit.Push(node);
                    }
                }
            }
        }
    }
}
```

6.3.2 Recherche en largeur

La recherche d'un chemin par un parcours en largeur ressemble beaucoup à celle par un parcours en profondeur. La classe **BreadthFirst** ressemble donc beaucoup à la précédente.

Le code est le même, à l'exception des lignes en gras qui permettent de remplacer la pile (Stack) par une file (Queue) et les méthodes Push/Pop par Enqueue et Dequeue (pour enfiler et enlever).

```
using System.Collections.Generic;

public class BreadthFirst : Algorithm
{
    public BreadthFirst(Graph _graph, IHM _ihm) : base(_graph, _ihm) {}

    protected override void Run()
    {
        // Création de la liste des noeuds non visités, et de la file
        List<Node> notVisitedNodes = graph.NodesList();
        Queue<Node> nodesToVisit = new Queue<Node>();
        nodesToVisit.Enqueue(graph.BeginningNode());
        notVisitedNodes.Remove(graph.BeginningNode());

        Node exitNode = graph.ExitNode();

        bool exitReached = false;
        // Boucle principale
        while (nodesToVisit.Count != 0 && !exitReached)
        {
            Node currentNode = nodesToVisit.Dequeue();
            if (currentNode.Equals(exitNode))
            {
                // On a fini
                exitReached = true;
            }
            else
            {
                // On ajoute les voisins
                foreach (Node node in graph.NodesList(currentNode))
                {
                    if (notVisitedNodes.Contains(node))
                    {
    notVisitedNodes.Remove(node);
                        node.Precursor = currentNode;
                        node.DistanceFromBegin =
currentNode.DistanceFromBegin +
```

```
graph.CostBetweenNodes(currentNode, node);
                                                   nodesToVisit.Enqueue(node);
                                        }
                                  }
                           }
                     }
              }
       }
```

6.3.3 Algorithme de Bellman-Ford

L'algorithme de Bellman-Ford est le premier à garantir le chemin le plus court. Il consiste à appliquer tous les arcs, et à mettre à jour les nœuds si on trouve un chemin plus court, et ce autant de fois que le nombre de nœuds (moins un). On s'arrête cependant si on ne peut plus améliorer les distances trouvées.

Il s'agit donc d'une grande boucle "tant que". À chaque arc, on regarde si le nœud atteint l'est alors de manière plus courte que jusqu'à présent. Si oui, on met à jour le prédécesseur et la distance au départ.

On rajoute une dernière itération à la fin, pour vérifier qu'il existe bien un chemin le plus court sinon on lève une exception.

```
using System;
using System.Collections.Generic;

public class BellmanFord : Algorithm
{
    public BellmanFord(Graph _graph, IHM _ihm) : base(_graph, _ihm) {}

    protected override void Run()
    {
          // Initialisation
          bool distanceChanged = true;
          int i = 0;
          List<Arc> arcsList = graph.ArcsList();

          // Boucle principale
          int nbLoopMax = graph.NodesCount() - 1;
          while (i < nbLoopMax && distanceChanged)
          {
                distanceChanged = false;
                foreach (Arc arc in arcsList)
                {
                       if (arc.FromNode.DistanceFromBegin +
```

```
arc.Cost < arc.ToNode.DistanceFromBegin)
                        {
                                arc.ToNode.DistanceFromBegin =
arc.FromNode.DistanceFromBegin + arc.Cost;
                                arc.ToNode.Precursor =
arc.FromNode;
                                distanceChanged = true;
                        }
                }
                i++;
        }

        // Test si boucle négative
        foreach (Arc arc in arcsList)
        {
                if (arc.FromNode.DistanceFromBegin + arc.Cost <
arc.ToNode.DistanceFromBegin)
                {
                        // Impossible de trouver un chemin
                        throw new Exception();
                }
        }
    }
}
```

6.3.4 Algorithme de Dijkstra

L'algorithme de Dijkstra s'applique, non sur les arcs, mais sur les nœuds. On choisit donc à chaque itération un nœud, et on applique tous les chemins sortants, en mettant à jour les nœuds adjacents pour lesquels on vient de trouver un chemin plus court qu'auparavant.

Pour choisir le nœud utilisé, on prend celui dont la distance à l'origine est la plus faible, après un parcours de notre liste.

```
using System.Collections.Generic;
using System.Linq;

public class Dijkstra : Algorithm
{
    public Dijkstra(Graph _graph, IHM _ihm) : base(_graph, _ihm) { }

    protected override void Run()
    {
            // Initialisation
            List<Node> nodesToVisit = graph.NodesList();
            bool exitReached = false;

            // Boucle principale
            while (nodesToVisit.Count != 0 && !exitReached)
            {
                    Node currentNode = nodesToVisit.FirstOrDefault();
                    foreach (Node newNode in nodesToVisit)
                    {
                            if (newNode.DistanceFromBegin <
currentNode.DistanceFromBegin)
                            {
                                    currentNode = newNode;
                            }
                    }

                    if (currentNode == graph.ExitNode())
                    {
                            exitReached = true;
                    }
                    else
                    {
                            List<Arc> arcsFromCurrentNode =
graph.ArcsList(currentNode);

                            foreach (Arc arc in arcsFromCurrentNode)
                            {
                                    if (arc.FromNode.DistanceFromBegin +
arc.Cost < arc.ToNode.DistanceFromBegin)
                                    {
                                            arc.ToNode.DistanceFromBegin =
arc.FromNode.DistanceFromBegin + arc.Cost;
                                            arc.ToNode.Precursor =
```

```
arc.FromNode;
                                        }
                                }

                                nodesToVisit.Remove(currentNode);
                        }
                }
        }
}
```

6.3.5 Algorithme A*

L'algorithme A* est presque le même que celui de Dijkstra. La différence est que l'on trie les nœuds non pas uniquement sur les distances au départ, mais sur la distance au départ ajoutée à la distance estimée à la sortie.

Tout d'abord un appel à la méthode permettant d'estimer ces distances est ajouté. On change ensuite la condition de notre parcours pour trouver l'élément ayant la distance totale la plus faible. Le reste du code est identique. La classe est remise entièrement ici, et les lignes modifiées sont en gras.

```
using System.Collections.Generic;
using System.Linq;

public class AStar : Algorithm
{
    public AStar(Graph _graph, IHM _ihm) : base(_graph, _ihm) { }

    protected override void Run()
    {
        // Initialisation
        graph.ComputeEstimatedDistanceToExit();
        List<Node> nodesToVisit = graph.NodesList();
        bool exitReached = false;

        // Boucle principale
        while (nodesToVisit.Count != 0 && !exitReached)
        {
            Node currentNode = nodesToVisit.FirstOrDefault();
            foreach (Node newNode in nodesToVisit)
            {
                if ((newNode.DistanceFromBegin +
newNode.EstimatedDistance) < (currentNode.DistanceFromBegin +
currentNode.EstimatedDistance))
                {
```

```
                                currentNode = newNode;
                            }
                }

                if (currentNode == graph.ExitNode())
                {
                        // Sortie trouvée : on a fini
                        exitReached = true;
                }
                else
                {
                        // On applique tous les arcs sortant du noeud
                        List<Arc> arcsFromCurrentNode =
graph.ArcsList(currentNode);

                        foreach (Arc arc in arcsFromCurrentNode)
                        {
                                if (arc.FromNode.DistanceFromBegin +
arc.Cost < arc.ToNode.DistanceFromBegin)
                                {
                                        arc.ToNode.DistanceFromBegin =
arc.FromNode.DistanceFromBegin + arc.Cost;
                                        arc.ToNode.Precursor =
arc.FromNode;
                                }
                        }

                        nodesToVisit.Remove(currentNode);
                    }
                }
            }
}
```

6.4 Application à la carte

Nous appliquons nos algorithmes au problème de recherche de chemins sur notre carte.

6.4.1 Tile et Tiletype

Dans ce problème, les nœuds du graphe sont les différentes cases. Elles sont appelées des tuiles et peuvent être de différents types : herbe, chemin, arbre, eau ou pont.

Nous commençons par définir les différents types **TileType** via une énumération, et une petite classe statique utilitaire qui permet de transformer un caractère en type. Cela permettra de rentrer une nouvelle carte plus simplement.

Nous avons choisi les conventions suivantes pour chaque type : les chemins sont représentés par des '.' (pour symboliser les graviers), l'arbre par '*' (le rond de l'arbre vu de dessus), l'herbe par ' ' (espace, car vide), l'eau par 'X' (infranchissable) et les ponts par '=' (symbolisant le pont).

Voici donc **TileType** et la classe **TileTypeConverter** :

```
using System;

public enum TileType { Grass, Tree, Water, Bridge, Path };

internal static class TileTypeConverter
{
    public static TileType TypeFromChar(Char _c)
    {
           switch (_c)
           {
                  case ' ':
                         return TileType.Grass;
                   case '*':
                         return TileType.Tree;
                  case 'X':
                         return TileType.Water;
                  case '=':
                         return TileType.Bridge;
                  case '.':
                         return TileType.Path;
           }
           throw new FormatException();
    }
}
```

Il faut maintenant coder les cases. On a ainsi une classe **Tile** qui hérite de la classe `Node`.

Nos tuiles possèdent, en plus des propriétés des nœuds, trois propriétés :

- Le type de tuile,
- La ligne dans la carte,
- La colonne.

Un constructeur débute cette classe.

```
using System;

internal class Tile : Node
{
    protected TileType tileType;

    internal int Row { get; set; }
    internal int Col { get; set; }

    public Tile(TileType _type, int _row, int _col)
    {
          tileType = _type;
          Row = _row;
          Col = _col;
    }
}
```

De plus, nous créons une méthode indiquant si on peut aller sur une case ou non. Pour cela, il suffit de regarder le type de la case : seuls les chemins, l'herbe et les ponts sont accessibles.

```
    internal bool IsValidPath()
    {
         return tileType.Equals(TileType.Bridge) ||
tileType.Equals(TileType.Grass) ||
tileType.Equals(TileType.Path);
    }
```

On rajoute une méthode nous indiquant le coût de la case. Les chemins ont un coût en points d'action de 1, l'herbe et les ponts de 2. On renvoie une distance infinie pour les cases non accessibles (arbres et eau).

```
    internal double Cost()
    {
            switch (tileType)
            {
                  case TileType.Path :
                        return 1;
                  case TileType.Bridge:
                  case TileType.Grass:
                        return 2;
                  default :
                        return double.PositiveInfinity;
            }
    }
```

On termine cette classe par une surcharge de la méthode `ToString()`, qui affiche les coordonnées de la case ainsi que son type :

```
    public override string ToString()
    {
            return "[" + Row + ";" + Col + ";" +
tileType.ToString() + "]";
    }
```

6.4.2 Implémentation de la carte

Les cases étant définies, on peut passer à la carte, représentée par la classe **Map** qui implémente l'interface `Graph`. Il s'agit de la classe la plus longue en termes de lignes de code.

On commence par définir les nouveaux attributs : la carte est représentée par un tableau de tuiles à deux dimensions (`tiles`). On garde aussi le nombre de lignes et de colonnes et les tuiles de départ et d'arrivée.

Pour des raisons d'optimisation, on conserve deux listes, à l'origine vides : la liste des nœuds et la liste des arcs.

```
using System;
using System.Collections.Generic;

public class Map : Graph
```

```
{
    Tile[,] tiles;
    int nbRows;
    int nbCols;

    Tile beginNode;
    Tile exitNode;

    List<Node> nodesList = null;
    List<Arc> arcsList = null;

    // Méthodes
}
```

Le constructeur attend en entrée une chaîne de caractères contenant le dessin de la carte façon "ASCII Art". On va donc initialiser le tableau de tuiles, et séparer chaque ligne puis chaque caractère et le remplir. On enregistre ensuite l'entrée et la sortie.

```
   public Map(String _map, int _beginRow, int _beginColumn, int
_exitRow, int _exitColumn)
   {
          // Création du tableau Tiles
          String[] mapRows = _map.Split(new char[] { '\n' });
          nbRows = mapRows.Length;
          nbCols = mapRows[0].Length;
          tiles = new Tile[nbRows,nbCols];

          // Remplissage
          for(int row = 0; row < nbRows; row++)
          {
                 for (int col = 0; col < nbCols; col++)
                 {
                        tiles[row, col] = new
Tile(TileTypeConverter.TypeFromChar(mapRows[row][col]), row, col);
                 }
          }

          // Entrée et sortie
          beginNode = tiles[_beginRow, _beginColumn];
          beginNode.DistanceFromBegin = beginNode.Cost();
          exitNode = tiles[_exitRow, _exitColumn];

          // Liste des noeuds et des arcs
```

```
            NodesList();
            ArcsList();
    }
```

Il faut ensuite implémenter toutes les méthodes de l'interface. On commence par les méthodes renvoyant les tuiles d'entrée et de sortie, qui ne font que renvoyer l'attribut correspondant.

```
    public Node BeginningNode()
    {
          return beginNode;
    }

    public Node ExitNode()
    {
          return exitNode;
    }
```

La méthode suivante doit renvoyer la liste de tous les nœuds. Si la liste n'a pas été créée lors d'un appel précédent, on va donc la créer. Pour cela, on parcourt toutes les cases du tableau, et on les ajoute à la liste.

```
    public List<Node> NodesList()
    {
            if (nodesList == null)
            {
                    nodesList = new List<Node>();
                    foreach (Node node in tiles)
                    {
                            nodesList.Add(node);
                    }
            }
            return nodesList;
    }
```

Il faut aussi une méthode qui ne renvoie que les nœuds adjacents au nœud donné en paramètre. Dans ce cas-là, on va tester les quatre voisins, et, s'ils sont atteignables, les ajouter à notre liste, puis la renvoyer. On doit aussi faire attention au bord de la carte.

```
    public List<Node> NodesList(Node _currentNode)
    {
            List<Node> nodesList = new List<Node>();

            int currentRow = ((Tile)_currentNode).Row;
            int currentCol = ((Tile)_currentNode).Col;

            // Renvoyer les voisins s'ils existent et sont accessibles
            if (currentRow - 1 >= 0 && tiles[currentRow - 1,
currentCol].IsValidPath())
            {
                    nodesList.Add(tiles[currentRow - 1, currentCol]);
            }
        if (currentCol - 1 >= 0 && tiles[currentRow, currentCol -
1].IsValidPath())
            {
                    nodesList.Add(tiles[currentRow, currentCol - 1]);
            }
            if (currentRow + 1 < nbRows && tiles[currentRow + 1,
currentCol].IsValidPath())
            {
                    nodesList.Add(tiles[currentRow + 1, currentCol]);
            }
            if (currentCol + 1 < nbCols && tiles[currentRow,
currentCol + 1].IsValidPath())
            {
                    nodesList.Add(tiles[currentRow, currentCol + 1]);
            }

            return nodesList;
    }
```

On complète la manipulation des nœuds par une méthode qui doit renvoyer le nombre de nœuds. On renvoie simplement le nombre de cases du tableau.

```
    public int NodesCount()
    {
            return nbRows * nbCols;
    }
```

Pour certains algorithmes comme Bellman-Ford, il faut aussi renvoyer non pas la liste des nœuds mais la liste des arcs. On va donc parcourir chaque case du tableau, puis pour chaque voisin regarder s'il est atteignable, et si oui, créer l'arc correspondant. On renvoie ensuite la liste créée. Pour éviter d'avoir à refaire ce parcours, on sauvegarde dans l'attribut correspondant cette liste, pour la renvoyer aux appels suivants.

```
   public List<Arc> ArcsList()
   {
          if (arcsList == null)
          {
                 arcsList = new List<Arc>();

                 for (int row = 0; row < nbRows; row++)
                 {
                        for (int col = 0; col < nbCols; col++)
                        {
                               if (tiles[row, col].IsValidPath())
                               {
                                      // Haut
                                      if (row - 1 >= 0 &&
tiles[row - 1, col].IsValidPath())
                                      {
                                             arcsList.Add(new
Arc(tiles[row, col], tiles[row - 1, col], tiles[row - 1,
col].Cost()))
                                      }
                                      // Gauche
                                      if (col - 1 >= 0 &&
tiles[row, col - 1].IsValidPath())
                                      {
                                             arcsList.Add(new
Arc(tiles[row, col], tiles[row, col - 1], tiles[row, col -
1].Cost()));
                                      }
                                      // Bas
                                      if (row + 1 < nbRows &&
tiles[row + 1, col].IsValidPath())
                                      {
                                             arcsList.Add(new
Arc(tiles[row, col], tiles[row + 1, col], tiles[row + 1,
col].Cost()));
                                      }
                                      // Droite
```

```
                                        if (col + 1 < nbCols &&
tiles[row, col + 1].IsValidPath())
                                        {
                                               arcsList.Add(new
Arc(tiles[row, col], tiles[row, col + 1], tiles[row, col +
1].Cost()));
                                        }
                                }
                        }
                }
            }
            return arcsList;
   }
```

On peut aussi renvoyer les arcs sortant d'un nœud donné. Le code est le même, mais uniquement pour un nœud.

```
   public List<Arc> ArcsList(Node _currentNode)
   {
          List<Arc> list = new List<Arc>();

          int currentRow = ((Tile)_currentNode).Row;
          int currentCol = ((Tile)_currentNode).Col;

          // Renvoyer les voisins
          if (currentRow - 1 >= 0 && tiles[currentRow - 1,
currentCol].IsValidPath())
          {
                 list.Add(new Arc(_currentNode, tiles[currentRow - 1,
currentCol], tiles[currentRow - 1, currentCol].Cost()));
          }
          if (currentCol - 1 >= 0 && tiles[currentRow, currentCol - 1].
IsValidPath())
          {
                 list.Add(new Arc(_currentNode, tiles[currentRow,
currentCol - 1], tiles[currentRow, currentCol - 1].Cost()));
          }
          if (currentRow + 1 < nbRows && tiles[currentRow + 1,
currentCol].IsValidPath())
          {
                 list.Add(new Arc(_currentNode, tiles[currentRow + 1,
currentCol], tiles[currentRow + 1, currentCol].Cost()));
          }
          if (currentCol + 1 < nbCols && tiles[currentRow,
currentCol + 1].IsValidPath())
          {
```

```
            list.Add(new Arc(_currentNode, tiles[currentRow,
currentCol + 1], tiles[currentRow, currentCol + 1].Cost()));
            }

            return list;
    }
```

La méthode suivante renvoie le coût pour aller d'une case à une autre. Dans notre cas, il s'agit simplement du coût de la case d'arrivée.

```
    public double CostBetweenNodes(Node _fromNode, Node _toNode)
    {
            return ((Tile)_toNode).Cost();
    }
```

La méthode `ReconstructPath` doit créer une chaîne contenant les différents nœuds parcourus pour aller du départ à l'arrivée. Il faut donc remonter les prédécesseurs au fur et à mesure, depuis la sortie jusqu'à l'arrivée (qui n'a pas de prédécesseur).

```
    public String ReconstructPath()
    {
            String resPath = "";
            Tile currentNode = exitNode;
            Tile prevNode = (Tile) exitNode.Precursor;
            while (prevNode != null)
            {
                  resPath = "-" + currentNode.ToString() + resPath;
                  currentNode = prevNode;
                  prevNode = (Tile) prevNode.Precursor;
            }
            resPath = currentNode.ToString() + resPath;
            return resPath;
    }
```

Pour certains algorithmes comme A*, il faut connaître la distance estimée à la sortie. On utilise la distance de Manhattan : il s'agit du nombre de cases horizontales ajouté au nombre de cases verticales pour relier la case en cours à la sortie. Comme on sous-estime toujours la distance, il s'agit d'une bonne heuristique pour A*.

```
    public void ComputeEstimatedDistanceToExit()
    {
           foreach (Tile tile in tiles)
           {
                   tile.EstimatedDistance = Math.Abs(exitNode.Row -
tile.Row) + Math.Abs(exitNode.Col - tile.Col);
           }
    }
```

La dernière méthode remet à `null` les listes d'arcs et de nœuds et réinitialise les distances et les précurseurs.

```
    public void Clear()
    {
           nodesList = null;
           arcsList = null;
           for (int row = 0; row < nbRows; row++)
           {
                  for (int col = 0; col < nbCols; col++)
                  {
                         tiles[row, col].DistanceFromBegin =
double.PositiveInfinity;
                         tiles[row, col].Precursor = null;
                  }
           }
           beginNode.DistanceFromBegin = beginNode.Cost();
    }
```

Notre code est maintenant terminé : le problème est entièrement codé.

6.4.3 Programme principal

La dernière étape est la création du problème principal. On commence donc par créer une nouvelle classe qui doit implémenter l'interface `IHM` et donc sa méthode `PrintResult()`. On affiche simplement la distance du chemin trouvé puis le parcours.

```
using System;
using PathfindingPCL;

class MainProgram : IHM
{
    static void Main(string[] _args)
    {
```

```
            // À compléter
        }

        // Autres méthodes

        public void PrintResult(String _path, double _distance)
        {
                Console.Out.WriteLine("Chemin (taille : " + _distance +
") : " + _path);
        }
}
```

On a besoin de lancer un algorithme au choix, à partir de son nom. Pour cela, on crée une première méthode RunAlgorithm. Celle-ci attend en paramètres le nom de l'algorithme, puis le graphe représentant le problème. On garde, pour faire un benchmark, la durée du traitement (calculée à partir des DateTime avant et après l'appel à la méthode Solve de l'algorithme). On affiche le nom de l'algorithme puis la durée en ms.

```
    private void RunAlgorithm(string _algoName, Graph _graph)
    {
            // Variables
            DateTime beginning;
            DateTime end;
            TimeSpan duration;
            Algorithm algo = null;

            // Création de l'algorithme
            switch (_algoName)
            {
                    case "Depth-First":
                            algo = new DepthFirst(_graph, this);
                            break;
                    case "Breadth-First" :
                            algo = new BreadthFirst(_graph, this);
                            break;
                    case "Bellman-Ford" :
                            algo = new BellmanFord(_graph, this);
                            break;
                    case "Dijkstra":
                            algo = new Dijkstra(_graph, this);
                            break;
                    case "A*":
```

```
                        algo = new AStar(_graph, this);
                        break;
            }

            // Résolution
            Console.Out.WriteLine("Algorithme : " + _algoName);
            beginning = DateTime.Now;
            algo.Solve();
            end = DateTime.Now;
            duration = end - beginning;
            Console.Out.WriteLine("Durée (ms) : " +
duration.TotalMilliseconds.ToString() + "\n");
    }
```

Comme on va vouloir comparer nos cinq algorithmes, on écrit une méthode permettant de les lancer l'un après l'autre.

```
    private void RunAllAlgorithms(Graph _graph)
    {
            // Résolution par une recherche en profondeur
            RunAlgorithm("Depth-First", _graph);

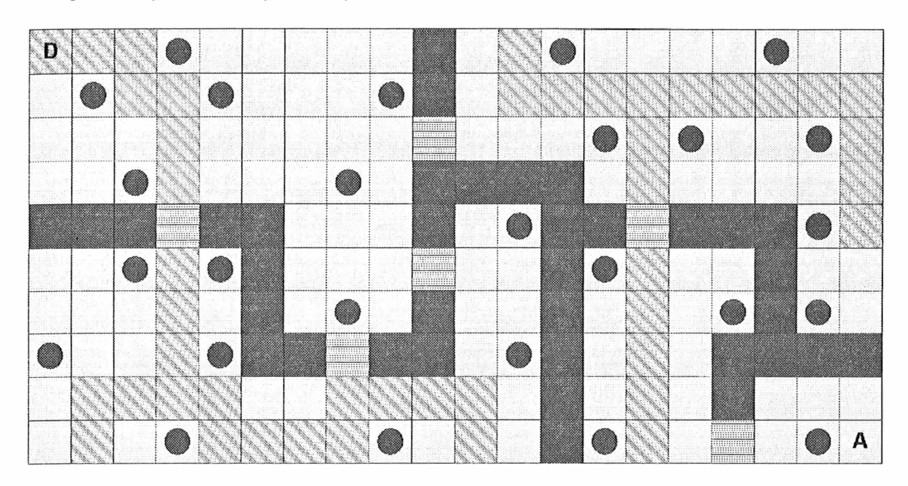
            // Résolution par une recherche en largeur
            RunAlgorithm("Breadth-First", _graph);

            // Résolution par Bellman-Ford
            RunAlgorithm("Bellman-Ford", _graph);

            // Résolution par Dijkstra
            RunAlgorithm("Dijkstra", _graph);

            // Résolution par A*
            RunAlgorithm("A*", _graph);
    }
```

On peut alors implémenter la méthode principale, `Run()`, qui va créer la carte, puis lancer les différents algorithmes. Pour cela, on crée la carte correspondant au problème présenté dans ce chapitre, et une deuxième carte plus longue et plus complexe que voici :



Les cartes seront représentées en ASCII, en remplaçant chaque case par le caractère correspondant.

Le code de cette méthode est le suivant :

```
private void Run()
{
       // 1ère carte
       String mapStr = "..  XX   .\n"
                             + "*.  *X  *.\n"
                             + " .  XX ...\n"
                             + " .* X *.* \n"
                             + " ...=...  \n"
                             + " .* X     \n"
                             + " .  XXX*  \n"
                             + " .  * =   \n"
                             + " .... XX  \n"
                             + "   *.  X* ";
       Map map = new Map(mapStr, 0, 0, 9, 9);
       RunAllAlgorithms(map);

       // 2ème carte
       mapStr = "...*     X .*    *  \n"
```

```
                + " *..*   *X .........\n"
                + "   .     =   *.*  *.\n*
                + "  *.   * XXXX .    .\n"
                + "XXX=XX   X *XX=XXX*.\n"
                + "  *.*X   =  X*.  X  \n"
                + "   . X * X  X . *X* \n"
                + "*  .*XX=XX *X . XXXX\n"
                + " ....  .... X . X   \n"
                + " . *....* . X*. = * ";
      map = new Map(mapStr, 0, 0, 9, 19);
      RunAllAlgorithms(map);
}
```

On termine par le contenu du `main`, qui se contente de créer un objet `MainProgram` et de lancer la méthode `Run` :

```
static void Main(string[] _args)
{
      MainProgram p = new MainProgram();
      p.Run();

      while (true) ;
}
```

Le programme est entièrement opérationnel, et il est facile de le tester sur une nouvelle carte. On pourrait imaginer que celles-ci soient chargées à partir d'un fichier texte par exemple.

6.5 Comparaison des performances

Nous allons maintenant comparer les performances de nos différents algorithmes.

Remarque

Les temps d'exécution dépendent fortement de la machine sur laquelle les algorithmes sont exécutés. Ils ne sont donc qu'une indication, car l'ordre des algorithmes du plus rapide au plus lent reste le même d'une machine à l'autre. De plus, les temps sont des moyennes effectuées sur 10 lancements sur un PC fixe avec un processeur Core 2 Duo E7500 et 8 Go de RAM.

Tout d'abord, les algorithmes de recherche en profondeur et en largeur trouvent des chemins, mais pas les plus optimisés. En effet, ils trouvent respectivement des chemins de 32 et 29 points d'action pour la première carte (optimum à 27) et 68 et 53 pour la deuxième carte (optimum à 49).

Par contre, dans les deux cas, l'algorithme A* trouve le chemin optimum. En effet, notre heuristique sous-estime (ou correspond à) la distance à la sortie, ce qui garantit de trouver le chemin le plus court. Ce n'est cependant pas forcément le cas selon les problèmes.

Au niveau des temps de traitement, on obtient les moyennes suivantes :

	Première carte	Deuxième carte
Profondeur	6.4 ms	0.3 ms
Largeur	0.9 ms	0.9 ms
Bellman-Ford	1.0 ms	0.6 ms
Dijkstra	2.6 ms	3.2 ms
A*	2.3 ms	3.0 ms

Malgré les 100 ou 200 cases et les centaines d'arcs, on remarque que tous nos algorithmes finissent en moins de 10 ms, ce qui est un temps très rapide.

On remarque aussi que sur la première carte, la recherche en profondeur n'est pas efficace du tout. En effet, nous n'avons pas choisi l'ordre de parcours des nœuds, et il semble que celui par défaut ne soit pas du tout adapté à ce problème. Au contraire, sur le deuxième problème, il est bien plus adapté et trouve la solution presque immédiatement.

La recherche en largeur, quant à elle, reste avec des temps équivalents : l'ordre de parcours n'est pas important. De plus, elle trouve des chemins globalement plus courts car nos arcs ont presque tous le même coût.

Pour les trois algorithmes de recherche des chemins les plus courts, on voit que Bellman-Ford est toujours le plus efficace. Ceci est dû au fait que nous avons finalement encore peu d'arcs et peu de nœuds, et que cet algorithme ne fait pas de tris sur les nœuds pour trouver le plus proche contrairement à Dijkstra et A*.

Dijkstra et A* fonctionnent sur le même principe, en cherchant le meilleur nœud (et c'est ce qui prend du temps). Cependant, selon les problèmes, l'un ou l'autre sera le plus efficace. Dans la première carte, il n'y a pas de pièges particuliers : le chemin le plus court suit globalement la ligne droite, A* est donc plus rapide. Sur la deuxième carte, au contraire, le chemin le plus proche de la ligne droite se retrouve bloqué juste avant l'arrivée par la rivière. Le chemin le plus court est un détour à l'origine, et c'est pourquoi l'algorithme A* est alors moins efficace que Dijkstra.

On peut aussi noter que le départ se trouve dans un angle. De cette façon, les algorithmes ne traitent qu'un quart de la zone entourant le point de départ, et c'est le bon quart (celui qui contient la sortie). Si le point de départ était au milieu d'une grande zone, Dijkstra ne serait pas efficace.

En conclusion :

– La recherche d'un chemin en largeur est globalement meilleure que la recherche en profondeur, car non sensible à l'ordre de parcours des nœuds.
– Bellman-Ford est le plus efficace sur les problèmes simples.
– Dijkstra et A* sont équivalents dans l'ensemble : s'il y a des pièges, comme dans le cas d'un labyrinthe par exemple, c'est Dijkstra le plus adapté ; au contraire, dans une grande zone dégagée avec quelques obstacles, c'est A* qu'il faut privilégier. De plus, si le point de départ est au centre de la zone, A* permet de limiter cette dernière grandement.

7. Domaines d'application

Ces algorithmes de recherche de chemins sont utilisés dans de nombreux domaines.

Le premier domaine est celui de la **recherche d'itinéraires**. Tous les GPS et les applications permettant d'aller d'un endroit à l'autre (en train, en bus, en métro, à pied...) utilisent des algorithmes de pathfinding. Ils prennent en compte la longueur du chemin ou son temps. Vue la complexité des cartes souvent utilisées (par exemple pour Google Maps), il est évident que les algorithmes doivent être optimisés, et privilégient les grands axes dès que possible. Le détail des algorithmes n'est bien évidemment pas communiqué.

Cette recherche de chemins se retrouve dans les **jeux vidéo**. Le but est alors de déplacer un personnage (contrôlé par le joueur ou représentant un ennemi) d'un endroit à un autre. Les cartes peuvent être très grandes, et le nombre de personnages important. Là encore, il faut optimiser les algorithmes utilisés. On peut cependant noter que c'est l'algorithme A* qui est majoritairement implémenté.

La **robotique** est un autre domaine friand de recherche d'itinéraires. Il s'agit alors d'amener un robot d'un point à un autre, le plus rapidement possible. Ces algorithmes sont généralement modifiés pour deux raisons. La première, c'est que l'environnement est fluctuant : si le robot évolue au milieu d'humains, ceux-ci seront en déplacement et vont barrer le chemin ou, au contraire, ouvrir d'autres routes au robot. Il doit donc recalculer en permanence le meilleur chemin. La deuxième raison, c'est que le robot n'a pas forcément une connaissance de l'ensemble de la carte. En effet, il ne connait que les zones qu'il a déjà visitées, et ces zones elles-mêmes peuvent changer (par exemple une porte fermée). La carte n'est donc pas stable dans le temps.

On utilise de nombreux algorithmes de pathfinding dans les réseaux, pour le **routage**. Internet en est un très bon exemple, avec de nombreux algorithmes permettant de décider de la meilleure façon de relier un client et un serveur ou d'envoyer des requêtes. Le protocole RIP (pour *Routing Information Protocol*) utilise ainsi Bellman-Ford, en envoyant toutes les 30 secondes les nouvelles routes (les machines pouvant à tout moment se connecter ou se déconnecter). La distance utilisée est simplement le nombre de sauts (qui correspond au nombre de machines dans la route). Le protocole OSPF (*Open Shortest Path First*), créé pour remplacer RIP, fonctionne lui avec l'algorithme de Dijkstra.

Enfin, la recherche de chemins peut être utilisée dans d'autres domaines. En **théorie des jeux**, on peut l'appliquer pour chercher un chemin allant de la position initiale à une position gagnante, ou au moins une position intéressante. En effet, il y a souvent trop de possibilités pour une recherche exhaustive.

C'est ainsi que fonctionnent beaucoup d'adversaires électroniques aux échecs : une recherche en largeur sur quelques niveaux va permettre de déterminer quel est le mouvement qui semble le plus avantageux. On considère qu'un humain peut tester trois niveaux de profondeur pour décider de son coup, là où Deep Blue, le superordinateur qui avait battu Kasparov, peut en tester près de 8.

Au final, tout problème pouvant s'exprimer sous la forme d'un graphe (comme la planification d'un projet, ou une procédure) peut utiliser un algorithme de pathfinding pour trouver le chemin le plus court, le plus rapide ou encore le plus économique. Les possibilités d'application sont donc très nombreuses.

8. Synthèse

La recherche de chemins, ou pathfinding, permet de relier des nœuds d'un graphe en utilisant des arcs prédéfinis. Ceux-ci sont associés à une longueur (ou coût). On peut ainsi chercher le chemin au coût le plus faible, que le coût soit en réalité un kilométrage, un temps ou encore un prix (par exemple l'essence consommée).

Plusieurs algorithmes existent, chacun ayant ses spécificités.

Lorsque l'on cherche principalement à savoir si un chemin existe, sans rechercher le plus court, on peut se tourner vers les algorithmes naïfs de recherche en profondeur ou en largeur. Si on sait globalement dans quelle direction aller, la recherche en profondeur peut être intéressante (à condition de bien préciser l'ordre de parcours des voisins).

La recherche en largeur donne généralement de meilleurs résultats et est surtout plus générique. Dans les deux cas, on avance de nœud en nœud et on mémorise les nouveaux nœuds adjacents découverts, que l'on visitera ultérieurement. Ce qui les différencie, c'est la structure utilisée pour stocker les voisins : une pile pour la recherche en profondeur et une file pour la recherche en largeur.

L'algorithme de Bellman-Ford permet, quant à lui, de trouver le chemin le plus court, et ce quel que soit le graphe. Il consiste à appliquer les arcs pour calculer les distances les plus courtes, à travers de nombreuses itérations. Il n'est pas rapide en termes de calculs mais est facile à implémenter, et peut être efficace car il ne nécessite pas de trier, ordonner ou rechercher des éléments dans une liste (contrairement aux algorithmes suivants).

L'algorithme de Dijkstra est plus "intelligent" que Bellman-Ford, car il applique une seule fois chacun des arcs, en choisissant à chaque fois le nœud le plus proche du départ non encore utilisé. De cette façon, les calculs ne sont effectués qu'une seule fois. Cependant, selon la façon dont l'algorithme est codé, il peut être moins efficace car il nécessite de retrouver le nœud le plus proche parmi une collection de nœuds.

Enfin, l'algorithme A*, très réputé principalement dans les jeux vidéo, utilise une heuristique permettant d'estimer la distance d'un nœud à la sortie. Il fonctionne comme Dijkstra, mais c'est la distance totale du chemin (c'est-à-dire la distance calculée depuis l'origine ajoutée à celle estimée jusqu'à l'arrivée) qui indique le nœud à prendre à l'étape suivante. Globalement, dans une carte dégagée, A* est plus efficace, alors que dans un labyrinthe, c'est Dijkstra qui est le plus performant.

Chapitre 4
Algorithmes génétiques

1. Présentation du chapitre

La nature a trouvé le moyen de résoudre certains problèmes en apparence insolubles. La vie est ainsi présente quasiment partout sur terre, des terres gelées aux fosses sous-marines (présentant des températures et pressions élevées) en passant par les airs.

Cette réussite s'explique par la puissance de l'évolution biologique. Elle permet d'adapter en permanence les différentes espèces aux milieux à coloniser.

Les informaticiens ont imaginé comment cette évolution pourrait être utilisée pour résoudre des problèmes complexes. C'est ainsi que les algorithmes génétiques sont apparus.

Dans une première partie, les principes sous-jacents à l'évolution biologique sont expliqués. Ils sont nécessaires pour comprendre le fonctionnement global et l'inspiration des algorithmes génétiques.

Ensuite sera présenté comment ceux-ci fonctionnent, tout d'abord de manière globale puis en revenant sur les principales étapes de leur fonctionnement, avec des bonnes pratiques et les pièges à éviter.

Ils peuvent être utilisés dans de nombreux domaines d'application présentés. Deux exemples d'implémentation sont ensuite proposés, en langage C#.

Ce chapitre se termine par une partie sur la coévolution, c'est-à-dire l'évolution conjointe de deux espèces (ou deux programmes en informatique), et une synthèse du chapitre.

2. Évolution biologique

Les algorithmes génétiques sont basés sur l'**évolution biologique**. S'il n'est pas nécessaire de comprendre tous les détails de celle-ci, il est cependant important de comprendre la source d'inspiration de ces algorithmes.

2.1 Le concept d'évolution

L'évolution biologique fut étudiée à partir de la fin du 18e siècle. En effet, les preuves de cette évolution s'accumulaient, et les scientifiques voulaient comprendre les phénomènes sous-jacents.

C'est au début du 19e siècle qu'est apparue la **paléontologie** (le terme est employé à partir de 1822), science qui s'intéresse aux fossiles et aux formes de vie aujourd'hui disparues. Les scientifiques trouvaient de nombreux squelettes et les classaient. Ceux-ci présentaient de fortes ressemblances entre eux, ou avec des formes de vie actuelles. Il semblait donc évident qu'il y avait eu une continuité, et que les espèces s'étaient progressivement modifiées au cours des millénaires.

De plus, les sociétés d'**élevage** étaient nombreuses. On savait depuis longtemps sélectionner les meilleurs individus d'un cheptel pour améliorer la production d'une espèce (comme le lait pour les vaches), ou simplement pour le plaisir. Les races de chiens, de chats ou de chevaux étaient ainsi nombreuses et fort différentes. Il était évident qu'un animal était proche de ses parents, bien que pas totalement identique à ceux-ci, et qu'en sélectionnant les bons parents, on pouvait créer de nouvelles races. Les caniches, les bergers allemands, les cockers et les labradors descendent tous du loup gris (Canis Lupus), domestiqué pendant la préhistoire. C'est l'intervention humaine qui a modelé toutes ces races.

Enfin, les grands découvreurs allaient d'îles en îles, et de nouvelles espèces étaient couramment découvertes. Il apparaissait que les individus situés sur des îles assez proches étaient eux-aussi proches physiquement. Au contraire, ils étaient beaucoup plus différents d'individus issus d'un autre continent. Les espèces avaient donc évolué de manière différente mais graduelle.

L'évolution biologique n'était donc plus un tabou au 19e siècle mais une réalité scientifique. Cependant, il restait à savoir comment cette évolution pouvait avoir eu lieu.

2.2 Les causes des mutations

Darwin (1809-1882) et Lamarck (1744-1829) s'opposèrent sur les raisons des modifications entre les parents et les descendants, appelées **mutations**.

D'après Darwin, un descendant était la moyenne de ses parents, mais parfois, aléatoirement, des différences apparaissaient. Seuls les individus les plus adaptés pouvant survivre et donc se reproduire, seules leurs modifications se transmettaient à leurs descendants. Les mutations qui n'étaient pas intéressantes au sens de la survie n'étaient donc pas propagées, et s'éteignaient. Au contraire, celles qui apportaient un avantage sélectif étaient conservées et se propageaient.

Lamarck, lui, avait proposé une autre théorie plusieurs décennies plus tôt : la transmission de caractères acquis pendant la vie de l'individu. Il pensait que ces variations étaient donc une réponse à un besoin physiologique interne. Ainsi, parce que les ancêtres des girafes avaient besoin de tendre leur cou de plus en plus haut pour attraper les feuilles des arbres, et que celui-ci devait s'allonger pendant la vie de l'animal de quelques centimètres ou millimètres, il s'était allongé de génération en génération, jusqu'à atteindre la hauteur actuelle.

De nombreuses expériences eurent lieu pour comprendre les causes de ces mutations et infirmer ou confirmer les hypothèses de Darwin et Lamarck. En coupant la queue de souris pendant de nombreuses générations, Weismann observa en 1888 qu'aucune ne finissait par naître sans, ce qui infirmait la transmission des caractères acquis.

La thèse de Darwin semblait confirmée (bien qu'officiellement, sa théorie ne fût reconnue qu'en 1930) : lors de la reproduction, des **mutations aléatoires** se produisent de temps en temps. Seuls les individus pour lesquels celles-ci sont bénéfiques vont devenir plus forts, plus résistants ou plus attirants et pourront ainsi se reproduire pour transmettre ce nouveau caractère. Souvent, et à tort, cette théorie est simplifiée par la tournure "la survie du plus fort", bien qu'en fait, il faudrait plutôt dire "la survie du plus adapté".

La façon dont apparaissaient ces mutations était par contre inconnue.

2.3 Le support de cette information : les facteurs

Une fois l'évolution reconnue, et les mutations réputées aléatoires, il restait à découvrir comment les informations génétiques étaient stockées dans un individu et comment elles pouvaient se transmettre.

C'est Gregor Mendel qui s'intéressa le premier à cette étude complexe. Pour cela, il commença par étudier l'hybridation de souris puis se concentra sur les petits pois à partir du milieu du 19e siècle.

Il choisit des plants de petits pois possédant sept caractéristiques bien distinctes, comme la forme ou la couleur de la graine, avec à chaque fois deux possibilités. Il croisa ensuite ces plants sur plusieurs générations et observa les plants obtenus. Il en déduisit alors ce que l'on nomme aujourd'hui les **lois de Mendel** (en 1866).

Celles-ci réfutent la théorie du mélange proposée par Darwin en complément de sa théorie de l'évolution (un enfant serait la moyenne de ses parents). En effet, à la première génération, tous les plants étaient "purs" : il n'y avait aucune moyenne de faite. Ainsi un croisement entre des plantes à fleurs blanches et des plantes à fleurs pourpres ne donnait pas naissance à des fleurs rosées mais à des fleurs pourpres.

Les lois suivantes indiquaient que chaque individu possédait des **facteurs** (au nombre de deux) ne contenant qu'une valeur possible pour un trait donné. Leur forme biologique exacte était alors inconnue. Lors de la reproduction, un seul facteur, choisi aléatoirement, était transmis aux descendants. Ainsi, les individus possédant deux facteurs pourpres croisés à des individus à deux facteurs blancs ne donnaient naissance qu'à des individus contenant un facteur pourpre et un blanc. Le facteur pourpre, **dominant**, donnait la couleur aux fleurs. Le facteur blanc était dit **récessif**.

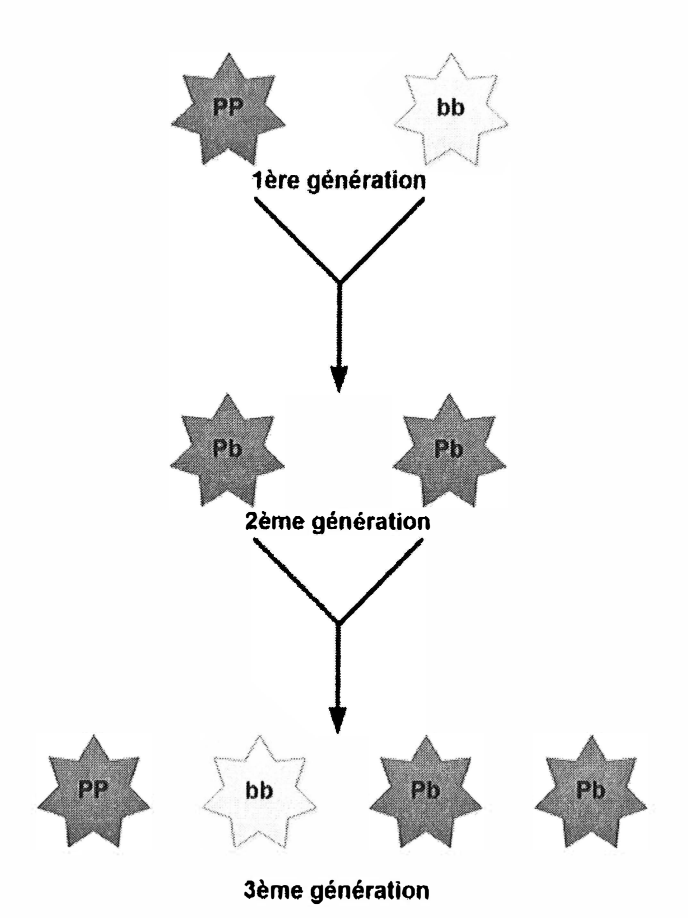
Son expérience peut donc être résumée sur la figure suivante. La première génération est constituée de plants "purs" : soit des fleurs pourpres depuis plusieurs générations qui possèdent donc deux facteurs P (pour Pourpre), soit des fleurs blanches qui possèdent deux facteurs b (pour blanc). La majuscule à Pourpre est une convention qui indique qu'il s'agit d'un facteur dominant.

Remarque

En biologie, on noterait le facteur blanc par p et non b. En effet, la convention consiste à choisir l'initiale du facteur dominant (ici P pour pourpre), et le mettre en majuscule pour la valeur (dite allèle) correspondante, et en minuscule pour l'allèle récessif (blanc pour nos fleurs).

La deuxième génération, qui est issue d'un croisement des plantes pures, n'a donc que des individus ayant hérité un facteur P d'un parent et b de l'autre. Ils sont donc tous Pb, et le pourpre étant dominant, ils apparaissent tous pourpres.

À la troisième génération, on a quatre possibilités. En effet, le premier parent peut donner un facteur P ou b, tout comme le deuxième. On a alors 25 % de PP qui sont pourpres, 25 % de bb qui sont blanches, et 50 % de Pb qui sont pourpres aussi (en fait 25 % de Pb et 25 % de bP, mais on note le facteur dominant en premier par convention).



On voit aussi ici pourquoi on dit dans le langage courant que certaines caractéristiques "sautent une génération" (on obtient une fleur blanche à la troisième génération bien qu'on n'en ait pas à la deuxième). Il s'agit en fait de traits récessifs.

Avec les lois de Mendel, les bases de la génétique sont fixées.

2.4 Des facteurs au code génétique

Les travaux de Mendel ne furent malheureusement pas connus de suite de la communauté scientifique. D'autres scientifiques continuèrent leurs travaux sur le stockage de l'information génétique et les découvertes s'enchaînèrent à grande vitesse.

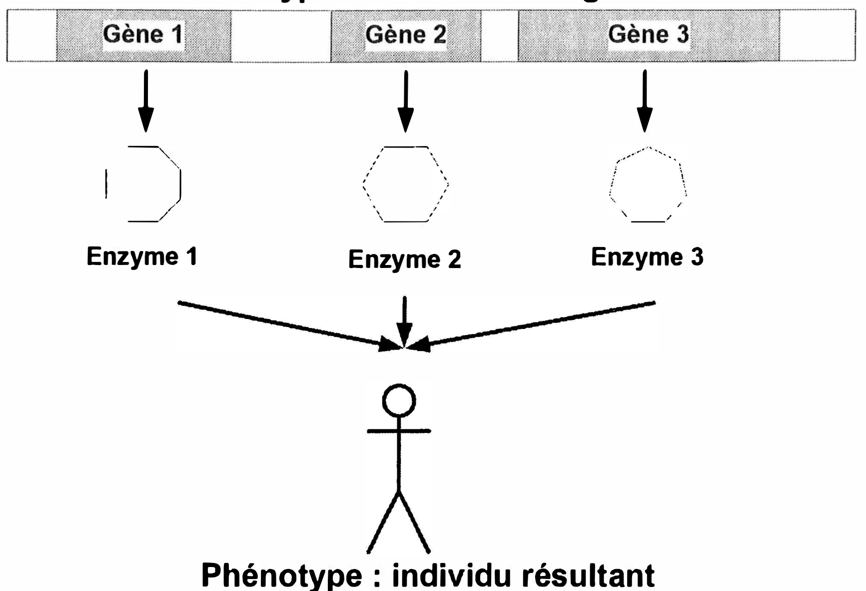
C'est ainsi que l'**ADN** fut isolé en 1869, puis les **chromosomes** en 1879 par Flemming. En 1900, les lois de Mendel furent redécouvertes par plusieurs chercheurs de manière indépendante. Il parut alors évident que c'était dans l'ADN des chromosomes que se situaient les facteurs de Mendel.

On parle dès 1909 de **gènes** au lieu de facteurs, et une première carte des gènes de la drosophile (son chromosome X) fut d'ailleurs proposée dès 1913. La structure de l'ADN en double hélice fut découverte en 1952 par Watson, Crick et Wilkins. Le **code génétique** fut admis dans les années 1960.

On comprend alors mieux le passage des gènes aux enzymes. Il se fait en deux temps : la **transcription**, qui transforme l'ADN en "ARN" (une sorte de négatif de l'ADN qui peut se déplacer jusqu'au lieu de production de l'enzyme), puis la **traduction**, qui permet de passer de l'ARN à la suite des acides aminés formant la protéine.

Les principes basiques de la **génétique** sont alors tous présents : un individu possède des chromosomes, contenant des gènes. Chaque gène correspond à une enzyme, grâce à un code qui indique la composition de celle-ci. L'ensemble des gènes d'un être vivant est appelé son **génotype**. Les interactions entre toutes les enzymes créent l'individu, appelé **phénotype**.

Voici un schéma récapitulatif :



Remarque

Les travaux ne se sont pas arrêtés dans les années 1960. Depuis, le processus a été mieux compris, car il est en fait beaucoup plus complexe. Par exemple, on sait maintenant qu'un seul gène, en fonction des caractéristiques de l'environnement, pourra permettre d'obtenir différentes enzymes, en gardant ou enlevant de manière sélective certaines zones de son code avant la création de l'enzyme. Les algorithmes génétiques n'utilisent cependant pas ces processus, et il est donc uniquement nécessaire de comprendre les principes basiques.

2.5 Le « cycle de la vie »

En résumé, un individu possède un ensemble de gènes (le génotype), présents à chaque fois en double exemplaire. La transcription puis la traduction permettent de transformer ces gènes en enzymes, qui vont réagir entre elles et créer l'être vivant (le phénotype).

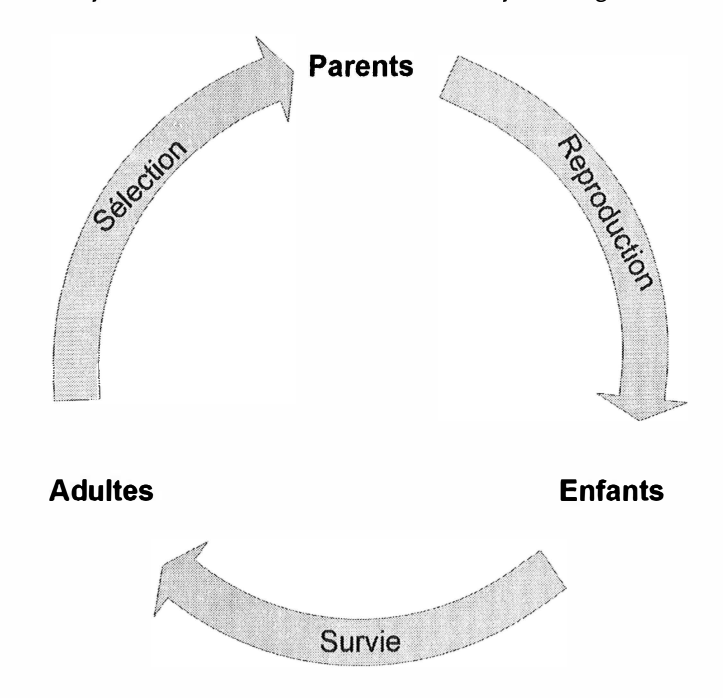
Lors de la reproduction, il va donner à son descendant la moitié de son capital génétique, qui sera mixé avec le capital génétique du deuxième parent. De plus, pendant ce processus, des mutations aléatoires peuvent se produire.

L'individu ainsi créé va ressembler à ses parents, tout en leur étant légèrement différent. Selon les mutations qu'il aura subies, il pourra être plus ou moins adapté que ses parents pour survivre dans son environnement.

S'il est plus adapté, il aura plus de chances de survivre, sera plus résistant, ou plus attrayant, et va donc pouvoir ensuite se reproduire. Au contraire, si les mutations qu'il a subies le rendent moins adapté, il aura plus de difficultés à survivre. Les causes sont nombreuses : mort prématurée de l'individu, faiblesse, mauvaise résistance aux maladies, difficultés à se nourrir ou se déplacer...

La **sélection naturelle** va donc avantager les mutations et les croisements d'individus intéressants pour la survie de l'espèce. Ceux-ci vont se disséminer dans la population et l'espèce va continuellement s'améliorer et s'adapter à son environnement.

On peut résumer ce "cercle de la vie" par la figure suivante :



3. Évolution artificielle

3.1 Principes

L'évolution biologique vue précédemment est dite "désincarnée" : en effet, les principes de reproduction, de survie ou de sélection ne précisent pas comment les informations doivent être stockées ou transmises, ni même ce qui doit évoluer.

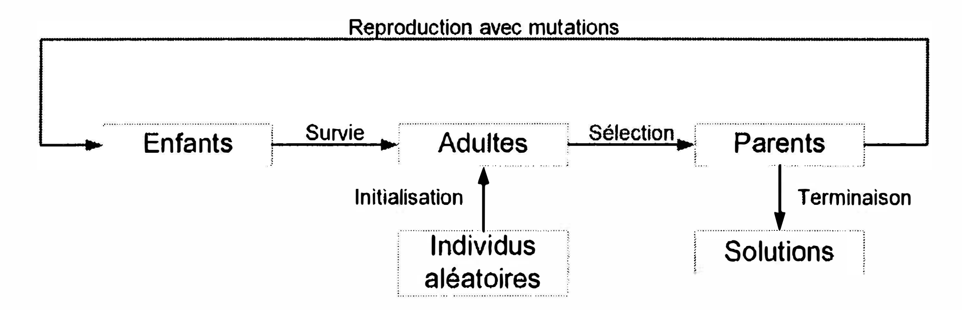
Les chercheurs de domaines très divers s'y sont donc intéressés, que ce soit l'économie, la sociologie, la musique... L'informatique n'est pas en reste, et cette évolution biologique peut être utilisée pour créer une évolution artificielle, permettant de résoudre des problèmes que des méthodes plus classiques ne permettent pas de résoudre.

Les algorithmes évolutionnaires vont donc partir d'une population de solutions potentielles à un problème. Chacune est évaluée, pour lui attribuer une note, appelée fitness. Plus la fitness d'une solution est forte et plus celle-ci est prometteuse.

Les meilleurs individus sont ensuite sélectionnés, et se reproduisent. Deux opérateurs artificiels sont alors utilisés : le croisement entre deux individus, appelé **crossover**, et des mutations aléatoires.

Une étape de survie s'applique alors pour créer la nouvelle génération d'individus.

Le processus est donc le suivant :



Plusieurs variantes sont apparues dans les années 60 de manière indépendante. Les **algorithmes génétiques** ont été mis au point par Holland, mais on parle aussi de **programmation évolutionnaire** (Fogel) ou de **stratégies d'évolution** (Rechenbert et Bäck). Quelques années plus tard sont apparues l'**évolution grammaticale** (Ryan, Collins et O'Neill) et la **programmation génétique** (Koza).

Tous ces algorithmes, souvent rassemblés sous le nom alors générique d'algorithmes génétiques, se basent sur ce principe d'évolution et cette boucle générationnelle. Les différences se font principalement au niveau des représentations et des opérateurs.

3.2 Vue d'ensemble du cycle

Il est important de comprendre les enjeux et les principes de chaque phase utilisée dans un processus d'évolution artificielle. Chacune est ensuite étudiée de manière plus approfondie.

3.2.1 Phases d'initialisation et de terminaison

Lors de la phase d'initialisation, une première population est créée. Pour la plupart des problèmes, on part de solutions aléatoires, qui sont donc en moyenne très peu adaptées au problème.

Si on connait déjà des solutions acceptables au problème, il est possible de directement les injecter lors de l'initialisation. Le processus complet est alors plus rapide, nécessitant moins de générations.

Il faut aussi définir un critère d'arrêt. Celui-ci permet de savoir à quel moment s'arrêter, pour donner à l'utilisateur les meilleures solutions trouvées. Ce critère peut porter sur un nombre de générations ou sur une fitness minimale à obtenir par les individus (un "score" à atteindre). Il peut aussi porter sur la découverte ou non de meilleures solutions (en étant par exemple du type "si pas de meilleure solution trouvée pendant X générations").

3.2.2 Phase de sélection

La sélection consiste à déterminer quels sont les individus qui méritent d'être choisis comme parents pour la génération suivante. Il faut qu'en proportion, les meilleurs parents se reproduisent plus que les parents à fitness plus basse mais chacun doit quand même avoir une probabilité non nulle de se reproduire.

En effet, c'est parfois en faisant muter ou en croisant des solutions en apparence "mauvaises" que l'on peut trouver une bonne solution à un problème.

3.2.3 Phase de reproduction avec mutations

Lors de la reproduction, on choisit pour chaque enfant de un à N parents. Les informations génétiques des différents parents sont mixées avec l'opérateur de crossover.

Une fois le mélange des parents effectué, on applique des mutations choisies aléatoirement au résultat, et dont le nombre dépend du taux de mutation de l'algorithme.

3.2.4 Phase de survie

Les enfants étant créés, il faut maintenant obtenir une nouvelle génération d'adultes qui peuvent ou non se reproduire. Si la solution la plus simple consiste à remplacer toute la génération des parents par la génération des enfants, il existe cependant plusieurs autres stratégies de survie.

On obtient donc à la fin de la survie une nouvelle population, et on peut reboucler tout le processus.

3.3 Convergence

La convergence vers la solution optimale est démontrée théoriquement. Cependant, rien ne précise le temps nécessaire pour converger vers cette solution, qui peut donc être supérieur à ce qui est acceptable.

Il est donc important de bien choisir les différents opérateurs (sélection, mutation, crossover et survie) et les représentations, au nombre de trois : des gènes, des individus et de la population.

4. Exemple du robinet

4.1 Présentation du problème

Nous allons appliquer ces principes à un premier exemple très simple : on souhaite utiliser un algorithme génétique pour connaître le bon réglage d'un mélangeur pour faire la vaisselle.

On souhaite donc obtenir une eau chaude mais non bouillante (pour protéger les mains de celui qui lave la vaisselle) et avec un débit important mais pas trop fort (sinon ça gicle et éclabousse).

Il faut donc déterminer le nombre de tours à faire pour le robinet d'eau chaude et pour celui d'eau froide. On a donc deux mesures à déterminer, qui sont nos deux gènes. Leurs valeurs sont des valeurs réelles, comprises entre 0 (robinet fermé) et 5 (robinet ouvert à fond).

4.2 Initialisation de l'algorithme

Pour commencer, on va initialiser une population de cinq individus (notés I1 à I5), choisis aléatoirement. On pourrait obtenir la population suivante (la première valeur correspond au robinet d'eau chaude) :

I1	1.5	0.2
I2	1.0	4.8
I3	1.7	1.3
I4	3.5	1.6
I5	3.6	3.2

4.3 Évaluation des individus

On doit ensuite évaluer chacun de ces individus, et lui affecter une note, qui est sa fitness (ou valeur d'adaptation). Ici, on va utiliser un cobaye humain, qui va tester les différentes possibilités et leur accorder une note entre 0 et 5.

			Fitness :
I1	1.5	0.2	2
I2	1.0	4.8	0.5
I3	1.7	1.3	4
I4	3.5	1.6	1
I5	3.6	3.2	2

On voit ainsi que la deuxième solution est très peu adaptée (note de 0.5) : il y a trop d'eau qui coule, et en plus elle est très froide. Au contraire, la solution I3 a une eau chaude (bien qu'encore un peu froide au goût du testeur) et un débit à peine trop fort. Elle obtient donc la bonne note de 4/5.

4.4 Reproduction avec mutations

On choisit ensuite les meilleurs individus qui se reproduiront. Les parents ayant les meilleurs scores sont ceux qui auront le plus d'enfants. On utilisera un crossover dans 50 % des cas (c'est-à-dire que les enfants auront 1 parent la moitié du temps, 2 l'autre moitié).

Le tirage donne donc les "couples" suivants pour créer les individus I6 à I10 :

– I6 : issu d'I3.

– I7 : issu du croisement d'I3 et d'I1.

– I8 : issu du croisement d'I4 et d'I5.

– I9 : issu du croisement d'I5 et d'I3.
– I10 : issu d'I1.

I3, qui était la meilleure solution, est donc réutilisé 3 fois sur 5 pour la création des descendants. Au contraire, la solution I2 (la moins bonne) n'est ici jamais utilisée. I4 sera utilisé une fois grâce à sa fitness de 1, et I1 et I5, avec leur fitness de 2, seront utilisés deux fois chacun.

Avant mutation, on obtient donc les descendants suivants. On peut voir que ceux qui n'ont qu'un parent sont pour le moment simplement des clones, alors que ceux ayant deux parents récupèrent le premier gène chez le premier parent et le deuxième chez le second : ils sont donc bien un mélange, au niveau génétique, de leurs ascendants.

I6	1.7	1.3
I7	1.7	0.2
I8	3.5	3.2
I9	3.6	1.3
I10	1.5	0.2

On applique des mutations avec un taux de 10 %, qui consistent à remplacer une valeur par une autre, tirée aléatoirement.

Remarque

Comme on a cinq individus créés qui contiennent chacun deux gènes, on a donc 10 valeurs qui pourraient muter. Statistiquement, un taux de mutation de 10 % va donc produire une mutation par génération, sur l'ensemble des individus.

Après mutation, voici donc notre nouvelle population (la valeur mutée apparaît sur un fond grisé) :

I6	1.7	1.3
I7	1.7	0.2
I8	3.5	3.2
I9	3.6	1.3
I10	1.5	1.1

4.5 Survie

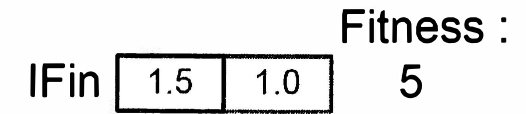
Enfin, on remplace tous les adultes actuels par la nouvelle génération créée. On évalue donc maintenant notre nouvelle population :

			Fitness :
I6	1.7	1.3	4
I7	1.7	0.2	2
I8	3.5	3.2	2
I9	3.6	1.3	1
I10	1.5	1.1	3

On peut remarquer qu'en une seule génération, les notes sont en moyenne bien plus élevées que lors de la première génération.

4.6 Suite du processus

On boucle ensuite tout le processus pour les générations suivantes : reproduction, remplacement de la population et évaluation jusqu'à ce qu'une solution donne entière satisfaction au testeur. On s'arrête alors et on garde cette solution, qui pourrait être la suivante (notée IFin pour indiquer que c'est l'individu conservé) :



5. Choix des représentations

Comme pour beaucoup de techniques d'intelligence artificielle, le choix des représentations est primordial pour limiter l'espace de recherche et pour le rendre le plus adapté possible à l'algorithme choisi.

5.1 Population et individus

La population contient une liste d'individus. C'est le langage informatique utilisé qui impose parfois la représentation de cette liste. Pour faciliter l'étape de reproduction, il est plus aisé de choisir une structure de données avec un accès direct à un individu, comme un tableau.

Les individus contiennent une liste de gènes. Là encore, le format exact de cette liste est en partie déterminé par le langage.

5.2 Gènes

La représentation des gènes est celle sur laquelle il faut passer le plus de temps. Traditionnellement, il s'agit d'une **liste ordonnée** de valeurs. Cela signifie que pour tous les individus, le premier gène a la même signification (dans notre exemple précédent, il s'agissait toujours du robinet d'eau chaude).

Dans certains cas, il peut cependant être plus adapté de choisir une représentation où la place des gènes est variable, en adaptant les opérateurs. Chaque gène contient alors le nom de la variable associée et la valeur.

De plus, il est important de bien réfléchir aux variables nécessaires pour résoudre le problème. Il est déconseillé d'avoir trop de valeurs à optimiser et il faut donc vérifier qu'il n'y a pas de variables redondantes, dont les valeurs pourraient être issues des autres.

Enfin, il est plus complexe pour un algorithme génétique de résoudre un problème dans lequel les différents gènes sont liés entre eux. Dans notre problème de robinet, nous cherchions à avoir la bonne température et la bonne pression à partir d'un mélangeur. Les deux variables sont donc fortement liées : si on augmente le nombre de tours du robinet d'eau chaude, on augmente de ce fait la pression et la température de l'eau. Conserver la pression mais changer la température demande à modifier les deux valeurs en même temps.

Au contraire, si nous avions eu un mitigeur (qui n'a donc qu'un robinet), on aurait pu plus facilement choisir la pression (il s'agit de la hauteur de ce dernier) et la température (selon son orientation à gauche ou à droite). De plus, il est alors facile de modifier la pression sans changer la température (et vice-versa). Le problème aurait été beaucoup plus simple.

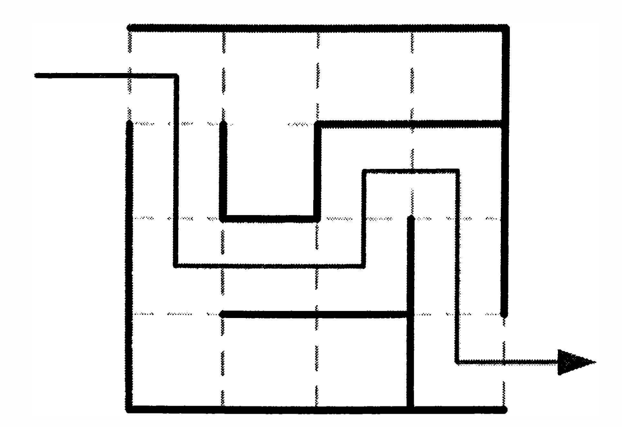
Remarque

Pour les humains aussi, résoudre un problème de type mitigeur est plus simple qu'un problème de type mélangeur.

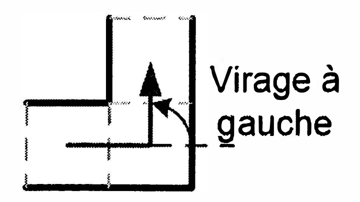
5.3 Cas d'un algorithme de résolution de labyrinthe

Il y a des cas où le choix de la représentation est plus complexe. Prenons le cas d'un labyrinthe : on veut trouver comment en sortir. Chaque individu représente donc une suite d'instructions (haut, bas, gauche, droite), et le but est d'arriver du début à la fin de ce dernier.

Voici un petit exemple de labyrinthe et sa solution :



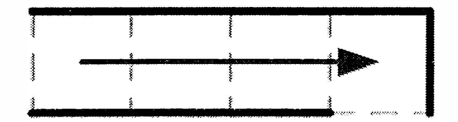
Il faut tout d'abord choisir si les directions sont absolues ou relatives. Dans le cas de directions relatives, chaque changement est dépendant de la direction actuelle. Ainsi, si on est en train d'aller vers la droite et qu'on a une direction "gauche", on va se retrouver à aller vers le haut. Le schéma suivant illustre ce principe :



Au contraire, dans le cas d'une direction absolue, on indique la direction voulue, dans l'exemple ci-dessus il s'agirait de "Haut".

Il faut ensuite choisir si les directions ont une valeur pour une seule case ou jusqu'au prochain croisement. S'il faut aller trois fois à droite pour atteindre le prochain carrefour, on aura trois fois la même instruction dans le premier cas, mais une seule fois dans le deuxième.

Ainsi, le chemin suivant s'indique [D, D, D] dans le premier cas, et seulement [D] dans le deuxième :



Voici donc les représentations du chemin permettant de sortir du labyrinthe donné en exemple dans les quatre cas discutés (instructions relatives ou absolues, et pour une case ou jusqu'à un changement). Chaque direction est représentée par son initiale (D pour droite, G pour gauche, H pour haut et B pour bas).

Portée → **Directions ↓**	**Une case**	**Plusieurs cases**
Absolues	[D,B, B, D, D, H, D, B, B, D]	[D, B, D, H, D, B, D]
Relatives	[D, D, H, G, H, G, D, D, H, G]	[D, D, G, G, D, D, G]

Dans le cas d'un algorithme génétique, des directions absolues sont plus intéressantes. En effet, cela découple la signification des différents gènes, le sens de l'un ne dépendant plus de celui des précédents. Une mutation du premier de "droite" à "haut", ne modifierait pas la suite du trajet. Il est aussi plus aisé pour un humain de donner les directions absolues en regardant le trajet qu'en réfléchissant à chaque fois à la direction dans laquelle on regarde à un moment donné.

De plus, dans le but de limiter au maximum le nombre de gènes, il semble plus opportun de conserver une direction jusqu'à un carrefour (ou un mur). Dans ce petit exemple, on passe ainsi de 10 à 7 gènes, soit un gain de 30 %.

6. Évaluation, sélection et survie

Nous allons maintenant nous intéresser aux opérateurs agissant sur la population et les individus, à savoir la sélection (qui commence par une évaluation) et la survie.

6.1 Choix de la fonction d'évaluation

Le choix de la **fonction d'évaluation** (ou fonction de fitness) est primordial, vu que c'est elle qui indique quel est le but à atteindre ou au moins dans quelle direction il faut aller.

Cette fonction peut être calculée à partir des données contenues dans les gènes via des fonctions mathématiques, mais ce n'est pas toujours le cas. En effet, elle peut aussi :

- Être donnée par un système externe qui "testerait" la solution proposée.
- Être attribuée par un humain qui jugerait de la qualité de la solution.
- Être obtenue après simulation de l'individu créé (qui peut alors être un programme informatique, un comportement dans un robot...).
- Ou être connue après un test réel par exemple lors de la création de pièces mécaniques.

La seule vraie contrainte est qu'elle permette de différencier les bons des mauvais individus.

Il faut cependant faire attention à ce qu'elle mesure bien le but recherché, car l'algorithme cherchant à la maximiser, il peut donner des résultats parfois surprenants...

Par exemple, pour le labyrinthe, si on note les individus par le nombre de cases parcourues, on risque de favoriser des individus qui vont faire des allers-retours entre deux cases à l'infini, ou des boucles. Si on mesure la distance à la sortie, on court le risque d'amener nos individus dans une impasse près de la sortie. Choisir une bonne fonction est donc parfois complexe.

Enfin, la fonction choisie doit être la plus continue possible (au sens mathématique) : elle ne doit pas présenter des paliers trop importants. En effet, il faut "guider" progressivement les individus vers les solutions les plus optimales. Elle doit donc proposer des valeurs graduellement croissantes, sans plateaux et sans brusques différences.

Ainsi, choisir une fonction de fitness pour notre labyrinthe qui octroierait 0 aux individus qui restent dans le labyrinthe et 50 à ceux qui en sortent ne pourrait pas mener à la convergence, et il faudrait compter sur le hasard pur pour trouver une solution valable. En effet, rien n'indiquerait alors à l'algorithme qu'il s'améliore et s'approche du but.

Remarque

Une analogie peut être faite avec le jeu du "froid ou chaud" dans lequel on indique à une personne où se trouve un objet. C'est grâce à une échelle graduelle allant du "gelé" au "tu vas bientôt bouillir", en passant par toutes les températures, que la personne peut trouver l'objet. Si on lui disait uniquement oui ou non, elle ne pourrait compter que sur la chance ou le parcours exhaustif des lieux pour trouver.

6.2 Opérateurs de sélection

Les parents peuvent être sélectionnés de diverses manières, déterministes ou stochastiques.

Une des solutions les plus courantes est d'utiliser une **roulette biaisée** : plus un individu est adapté, et plus il aura une grande part sur la roue. Les individus suivants ont donc une part de plus en plus petite. Un tirage au sort indique alors quel est l'individu choisi.

Statistiquement, ceux ayant les fitness les plus élevées auront le plus d'enfants, mais tous ont au moins une chance de se reproduire, même si elle reste faible.

La part de la roulette de chaque individu peut être déterminée par le rang de celui-ci, le premier ayant toujours la même part par rapport au deuxième, ou par sa fitness. Dans ce dernier cas, les proportions changent à chaque génération et un individu beaucoup plus adapté que les autres de sa population aura

beaucoup plus de descendants, pour transmettre rapidement ses gènes. Au contraire, dans une population uniforme où les différences de fitness sont faibles, la roulette donnera à peu près à chaque individu la même chance de se reproduire.

La deuxième solution, après la roulette, est d'utiliser le **tournoi** : deux individus sont choisis au hasard, et c'est le plus adapté qui se reproduira. Les individus les plus adaptés gagneront donc plus souvent les tournois et se reproduiront plus que les individus peu adaptés, mais là encore, tous ont des chances de se reproduire (à l'exception de l'individu le moins adapté qui perdra tous ses tournois).

La troisième solution est d'utiliser une **méthode déterministe** : on calcule le nombre de descendants de chaque individu sans tirage au sort, par une formule mathématique ou des règles choisies en amont. On peut par exemple décider pour une population de 15 individus que le meilleur individu aura toujours 5 enfants, que le deuxième en aura 4, jusqu'au cinquième qui aura 1 enfant, les suivants ne se reproduisant pas.

Enfin, on peut rajouter un caractère **élitiste** à cette sélection, qui permet de conserver le meilleur individu qui est donc cloné pour créer son propre descendant, sans mutation. De cette façon, on s'assure de ne jamais perdre une bonne solution.

Il n'existe cependant aucune façon de connaître l'opérateur de sélection le plus adapté à un cas donné. Il faut donc parfois faire des tests empiriques en changeant la solution retenue. On peut cependant noter que l'élitisme, à part pour des problèmes critiques et particuliers, n'est pas souvent utile, voire se trouve être néfaste à l'algorithme sur des problèmes complexes. En effet, en gardant systématiquement une bonne réponse, on peut passer à côté d'une meilleure réponse qui en est éloignée.

6.3 Opérateurs de survie

Lorsque les descendants sont créés, on se retrouve avec l'ancienne population composée entre autres des parents, et la nouvelle génération. Il ne faut cependant garder qu'une seule population.

La solution la plus simple consiste en un **remplacement** total des adultes par les enfants. Tous les individus survivent donc pendant une génération.

D'autres solutions sont cependant possibles, comme des **tournois** entre des individus des deux populations, soit en opposant systématiquement un individu de chaque génération, soit en les tirant au sort dans la population complète. Tous les individus qui ont été sélectionnés pour faire partie de la nouvelle population ne peuvent plus rentrer dans d'autres tournois.

Là encore on peut aussi utiliser des **méthodes déterministes**, consistant à choisir les meilleurs individus sur les deux générations.

On conseille cependant de choisir une méthode stochastique et une déterministe pour les opérateurs de sélection et de survie. On peut ainsi choisir une roulette biaisée (stochastique) pour la sélection et un remplacement (déterministe) pour la survie.

7. Reproduction : crossover et mutation

Nous allons maintenant nous intéresser aux opérateurs qui agissent sur les gènes et non plus sur l'individu ou la population. Ceux-ci sont au nombre de deux : le crossover, qui permet de mélanger les informations génétiques de différents parents, et la mutation, qui permet d'introduire de la variabilité aléatoirement.

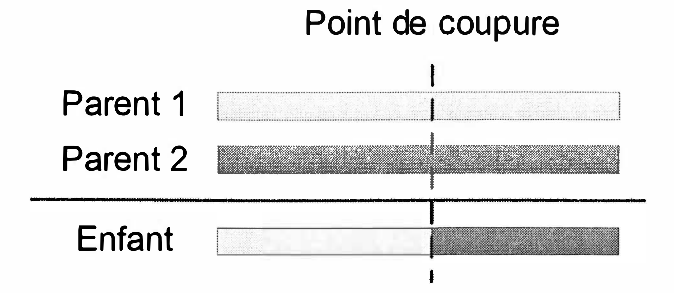
7.1 Crossover

Le **crossover**, parfois appelé "opérateur de croisement", permet de créer un nouveau descendant à partir de ses deux ascendants, en mixant les informations génétiques.

Remarque

Un descendant peut n'avoir qu'un seul parent (il n'y a alors pas de crossover), en avoir deux (c'est le cas le plus classique) ou plus. Là encore c'est au concepteur de choisir. Nous ne parlerons ici que des crossovers entre deux parents, mais les principes énoncés peuvent facilement être généralisés pour trois ou plus.

Le crossover le plus courant consiste à prendre un point de coupure dans le génome. Tous les gènes situés avant ce point viennent du premier parent, et ceux situés après, du deuxième. C'est aussi l'opérateur le plus proche de la réalité biologique. Il est dit **discret**, car il garde les valeurs telles quelles.



On peut cependant imaginer des variantes, en faisant par exemple la moyenne des parents pour chaque gène. Le crossover est alors dit **continu**. Mathématiquement, il consiste à prendre le milieu du segment représenté par les deux parents.

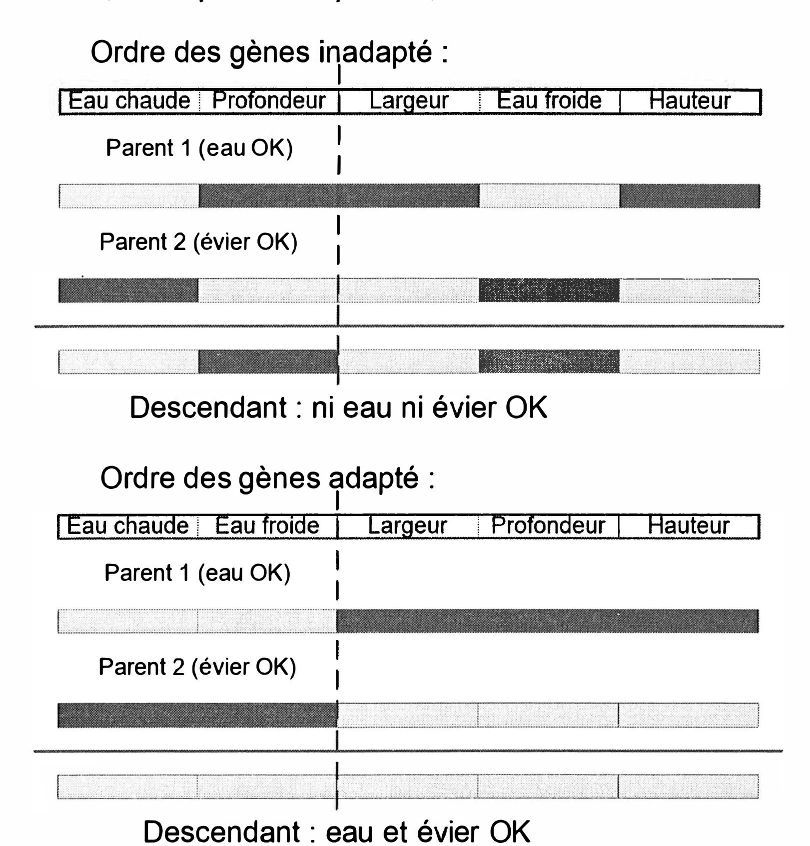
Cet opérateur n'est pas forcément utilisé pour chaque reproduction : il est possible de créer des descendants avec un seul parent, sans avoir besoin de crossover. Il faut donc déterminer le **taux de crossover** de l'algorithme, généralement supérieur à 50 %. Là encore, c'est l'expérience et le problème qui guideront les choix.

Le crossover n'est cependant pas valide et produit presque toujours des mauvais descendants dans deux cas :

- Si les gènes liés sémantiquement sont éloignés,
- Si les gènes sont contraints par le reste du génome.

Pour le premier cas, imaginons que nous complétions notre problème d'eau de vaisselle avec un autre sur la taille de l'évier. Les gènes correspondant aux deux robinets (eau chaude et eau froide) devraient rester proches sur le génome et ne pas être séparés par la largeur, la profondeur ou la hauteur de l'évier. De même, ces trois dernières variables doivent rester proches. En effet, si on les sépare, un individu ayant une bonne solution pour l'eau ne pourra pas se mixer avantageusement avec un individu ayant une bonne solution pour l'évier.

Voici donc ce que l'on pourrait obtenir avec un ordre inadapté puis avec un ordre adapté. On voit que dans le deuxième cas, on peut résoudre notre problème, alors que dans le premier, l'individu créé est mauvais.



Il est donc important de bien choisir l'ordre des variables, pour aider l'algorithme et optimiser les croisements.

Dans le deuxième cas, imaginons un problème de voyageur de commerce. Il s'agit d'un problème combinatoire classique, dans lequel un individu doit visiter plusieurs villes en faisant le moins de kilomètres possible. Il faut donc optimiser son chemin. Le génome d'un tel individu pourrait correspondre à la suite des villes à visiter. Si on a cinq villes nommées A à E, on voit bien que le croisement des deux individus suivants n'aurait pas de sens : certaines villes seraient visitées deux fois (C et E) et d'autres aucune (B et D) !

Parent 1 :	A	E	C	D	B
Parent 2 :	B	D	A	C	E
Enfant :	A	E	C	C	E

Dans ce cas, il faut adapter l'opérateur. Il peut par exemple prendre les N premières villes du premier individu, puis prendre les villes manquantes dans l'ordre du deuxième individu.

Parent 1 :	A	E	C	D	B
Parent 2 :	B	D	~~A~~	~~C~~	~~E~~
Enfant :	A	E	C	B	D

Le crossover est donc lié à la représentation choisie pour les gènes. Si la version discrète est la plus courante, elle n'est donc pas à appliquer sans vérifier qu'elle correspond à notre problème.

7.2 Mutation

Le deuxième opérateur local est l'opérateur de **mutation**. Il a pour but d'introduire de la nouveauté au sein de la population, pour permettre la découverte de nouvelles solutions potentielles.

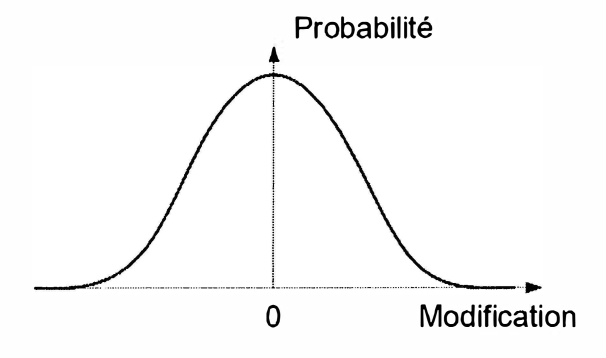
Il consiste donc à choisir aléatoirement certains gènes. La probabilité qu'un gène soit touché par une mutation s'appelle le **taux de mutation**. S'il est trop élevé, les bonnes solutions risquent fort de disparaître. Trop faible, il ne permet pas de trouver de nouvelles solutions rapidement. Il faut donc trouver le bon compromis.

Là encore, en fonction de la taille de la population, du nombre de gènes ou du problème, on choisira des taux différents. Un bon départ consiste cependant à partir d'un taux de 5 %, et à l'adapter ensuite selon les besoins.

Au niveau de son effet, la mutation peut suivre une **distribution uniforme** : dans ce cas, la nouvelle valeur du gène sera tirée au sort dans tout l'espace possible. Ainsi, pour notre problème de robinet, une nouvelle valeur serait tirée au sort entre 0 (robinet fermé) et 5 (robinet complètement ouvert).

L'opérateur peut aussi, et c'est généralement plus efficace, modifier la valeur actuelle par une valeur proche. Il suit alors une **distribution normale** (aussi appelée "courbe en cloche"), centrée sur 0, et permettant d'obtenir la modification à appliquer à la version actuelle.

Voici un exemple de distribution normale :



Ce type de mutation permet, surtout si l'espace de recherche est grand, de se déplacer graduellement dans celui-ci.

De plus, dans certains cas, les mutations peuvent consister à :

- Ajouter un gène ou en supprimer un, à la fin ou en milieu de chromosome.
- Dupliquer un gène (en particulier dans le cas d'un nombre variable de gènes).
- Échanger la place de deux gènes (par exemple pour le voyageur de commerce, en changeant l'ordre de deux villes).
- Etc.

Il faut donc s'adapter au problème à résoudre, comme pour les autres opérateurs.

8. Domaines d'application

L'équipe de John Holland au sein de l'université du Michigan a commencé à travailler sur les algorithmes génétiques dans les années 60. Cependant, cette technologie n'a commencé à se faire connaître qu'à partir de 1975 avec la publication de son livre.

Les algorithmes évolutionnaires en général ont alors commencé à toucher de nombreux domaines. Pour qu'ils soient efficaces, il suffit de répondre à quelques contraintes :

- Le nombre de solutions potentielles doit être très grand.
- Il n'y a pas de méthode exacte permettant d'obtenir une solution.
- Une méthode presque optimale est acceptable.
- On peut évaluer la qualité d'une solution potentielle.

Si ces quatre contraintes sont vérifiées, alors un algorithme génétique peut s'avérer être une bonne solution pour trouver une réponse au problème qui, bien qu'elle ne puisse être garantie comme la meilleure, sera en tout cas acceptable, et ce dans un temps raisonnable.

On les retrouve en premier dans les domaines de l'**ingénierie** et du **design**. En effet, il est aujourd'hui de plus en plus difficile de créer des pièces répondant aux contraintes et minimisant ou maximisant certaines caractéristiques (moins de matière première, consommation plus faible, puissance plus importante, meilleure résistance...). Ainsi, les carrosseries de voiture peuvent être créées par algorithme génétique pour les rendre plus aérodynamiques.

Leur deuxième grand domaine d'application est la **logistique**. On retrouve alors ici les problèmes de type "voyageur de commerce", mais aussi des problèmes d'optimisation sous contraintes. Ils peuvent servir à aménager au mieux un entrepôt pour limiter les trajets, à choisir les horaires de bus, de trains ou d'avions les plus adaptés, à améliorer un réseau informatique en réaménageant les différents nœuds pour limiter les goulots d'étranglement, à créer des emplois du temps...

Les laboratoires de **biologie** et **biochimie** sont aussi de gros consommateurs d'algorithmes génétiques. En effet, ceux-ci permettent d'aider au séquençage d'ADN, à la découverte de nouvelles protéines, au calcul de la forme repliée d'une molécule... Les exemples sont nombreux en particulier dans les laboratoires de recherche comme le laboratoire de bio-informatique comparée de Notredame en Espagne.

La **finance** est un domaine très complexe. Là encore cette technique peut aider, pour améliorer des prévisions boursières, gérer des portefeuilles, ou optimiser ses investissements. Des livres entiers sont d'ailleurs consacrés à l'utilisation des algorithmes génétiques dans ce domaine, comme "Genetic Algorithms and Investment Strategies" aux éditions Wiley.

Ils apportent aussi une aide à la **création**. Une utilisation commerciale est la création de nouveaux packagings pour des produits divers et variés, comme Staples pour ses emballages de ramettes de papier ou Danone pour le lancement du produit Activia aux États-Unis. Hors marketing, ils servent aussi à créer de la musique ou des images. Une équipe de l'INRIA (Institut National de Recherche en Informatique et en Automatique) avait même créé une gamme de foulards dont les dessins étaient obtenus par un algorithme génétique.

Enfin, on peut les trouver dans des domaines comme les **jeux vidéo** (pour créer des comportements pour les adversaires, des niveaux ou des personnages), ou dans des domaines plus originaux comme la **forensique**. Il existe ainsi un programme qui permet d'aider un témoin à créer un portrait-robot d'un criminel. Celui-ci propose plusieurs visages, et le témoin choisit à chaque génération ceux qui sont le plus proches de ses souvenirs, ce qui est plus simple que de choisir chaque partie du visage séparément.

9. Implémentation d'un algorithme génétique

Nous allons maintenant nous intéresser à l'implémentation en C# d'un algorithme génétique générique qui est ici utilisé pour résoudre deux problèmes abordés précédemment :

- Le voyageur de commerce, qui consiste à trouver la route la plus courte pour relier un ensemble de villes.
- Le labyrinthe, en donnant la suite d'instructions à suivre pour aller de l'entrée à la sortie.

Le code proposé ici et disponible en téléchargement est compatible avec .NET 4 et supérieur, Silverlight 5, Windows Phone 8 et supérieur, et les applications Windows Store pour Windows 8 et supérieur. Le programme contenant le `main` est une application console pour Windows.

9.1 Implémentation générique d'un algorithme

9.1.1 Spécifications

Nous voulons coder un moteur générique pour un algorithme génétique, qui est ensuite appliqué à deux problèmes différents, en écrivant le moins de code possible pour passer de l'un à l'autre.

Il est donc important de bien fixer les besoins. Le processus évolutionnaire en lui-même, le cœur du système, s'occupe d'initialiser la population, puis lance l'évaluation, la sélection des parents et la création des descendants et enfin la survie. On reboucle ensuite sur l'évaluation, et ce jusqu'à ce qu'un critère d'arrêt soit atteint.

On va donc définir deux critères d'arrêt possibles : on a atteint la fitness que l'on voulait ou on a atteint le nombre maximum de générations. Dans les deux problèmes, il s'agit de minimiser la fonction d'évaluation : le nombre de kilomètres pour le voyageur de commerce ou la distance à la sortie pour le labyrinthe. On fixe donc une fitness minimale à atteindre.

Remarque

On pourrait tout à fait adapter l'algorithme pour lui permettre de maximiser la valeur d'adaptation, mais comme nos deux problèmes cherchent à minimiser la fitness, nous ne le ferons pas ici.

Les différents paramètres de l'algorithme sont définis dans une classe à part. Nous avons ensuite des interfaces ou classes abstraites pour les individus et les gènes. En effet, ce sont les deux seules classes qu'il faut redéfinir pour chaque cas. Pour le problème de sortie du labyrinthe, nous avons besoin d'avoir des génomes de taille variable (la liste des instructions), l'algorithme doit donc permettre de le gérer.

Pour alléger le cœur du système de la gestion du cas à résoudre, nous déportons dans une fabrique la création des individus et l'initialisation de leur environnement.

Enfin, nous définissons une interface pour le programme principal. En effet, dans notre cas, nous allons faire des sorties dans la console, mais nous pourrions facilement adapter notre programme pour ajouter des lignes dans un tableau ou faire un affichage graphique des meilleurs individus.

9.1.2 Paramètres

Il faut commencer par définir une classe statique `Parameters` qui contient tous les paramètres. Ceux-ci sont initialisés avec une valeur par défaut, qui est la valeur généralement conseillée comme point de départ, et ils sont accessibles depuis les autres classes.

Nous définissons en premier les paramètres concernant toute la population et l'algorithme en général :

– Le nombre d'individus par génération, nommé `individualsNb` et initialisé à 20.
– Le nombre maximal de générations, nommé `generationsMaxNb` et initialisé à 50.
– Le nombre initial de gènes si le génome est de taille variable, nommé `initialGenesNb` et initialisé à 10.
– La fitness à atteindre, nommée `minFitness`, et initialisée à 0.

On définit ensuite les différents taux utilisés lors de la reproduction : le taux de mutations (`mutationsRate`) à 0.1, le taux d'ajout de gènes (`mutationAddRate`) à 0.2, le taux de suppression de gènes (`mutationDeleteRate`) à 0.1 et le taux de crossover (`crossoverRate`) à 0.6.

On termine cette classe par la création d'un générateur aléatoire qui pourra ensuite être utilisé dans tout le code, sans avoir à être recréé. Au cas où on souhaite pouvoir reproduire les résultats, il suffit d'indiquer une graine au générateur aléatoire.

Le code complet de cette classe est donc le suivant :

```
using System;

public static class Parameters
{
    public static int individualsNb = 20;
    public static int generationsMaxNb = 50;
    public static int initialGenesNb = 10;
    public static int minFitness = 0;

    public static double mutationsRate = 0.10;
    public static double mutationAddRate = 0.20;
    public static double mutationDeleteRate = 0.10;
   public static double crossoverRate = 0.60;

    public static Random randomGenerator = new Random();
}
```

9.1.3 Individus et gènes

Il faut ensuite définir une interface pour nos gènes, nommée **IGene**. Elle ne contient qu'une fonction de mutation. Rien n'est indiqué sur la façon de gérer les gènes, celle-ci dépendant du problème à résoudre.

```
using System;

internal interface Igene
{
    void Mutate();
}
```

Les individus, quant à eux, sont implémentés via la classe abstraite **Individual**. On y trouve le code nécessaire pour récupérer la fitness d'un individu ou son génome.

On rajoute ensuite deux fonctions abstraites pures (qu'il faut donc obligatoirement redéfinir dans les descendants) : une pour évaluer l'individu et une pour le faire muter.

Enfin, une fonction `ToString` est utilisée pour les affichages. On se contente d'afficher la valeur de fitness, suivie du génome. Pour cela, on utilise la fonction `Join()` qui permet de transformer une liste en une chaîne avec le délimiteur choisi.

On obtient donc le code suivant :

```
using System;
using System.Collections.Generic;

public abstract class Individual
{
    protected double fitness = -1;
    public double Fitness
    {
          get {
                 return fitness;
          }
    }

    internal List<IGene> genome;

    internal abstract void Mutate();
```

```
    internal abstract double Evaluate();

    public override string ToString()
    {
        String gen = fitness + " : ";
        gen += String.Join(" - ", genome);
        return gen;
    }
}
```

Pour chaque problème, une classe fille hérite de cette classe `Individual`. C'est le rôle d'une autre classe de choisir quelle classe fille instancier en fonction du besoin. Si on laisse ce choix dans le cœur de l'algorithme, on perd en généralité.

On utilise donc une fabrique d'individus, qui est un singleton. De cette façon, elle est accessible depuis tout le code, mais une seule instance est créée à chaque lancement du code.

Le code de base de la fabrique **`IndividualFactory`** est donc le suivant :

```
using System;

internal class IndividualFactory
{
    private static IndividualFactory instance;

    private IndividualFactory() { }

    public static IndividualFactory getInstance()
    {
        if (instance == null)
        {
            instance = new IndividualFactory();
        }
        return instance;
    }
}
```

Celle-ci permet d'initialiser l'environnement de vie des individus via une méthode `Init`, d'obtenir l'individu voulu, en le créant aléatoirement, ou à partir d'un ou deux parents grâce à trois méthodes `getIndividual()` qui prennent en paramètre le problème à résoudre sous la forme d'une chaîne, puis les parents potentiels.

```
    internal void Init(string type)
    {
            // Code à mettre ultérieurement ici
    }

    public Individual getIndividual(String type) {
            Individual ind = null;
            // Code à mettre ultérieurement ici
            return ind;
    }

    public Individual getIndividual(String type, Individual father)
    {
            Individual ind = null;
            // Code à mettre ultérieurement ici
            return ind;
    }

    public Individual getIndividual(String type, Individual father,
Individual mother)
    {
            Individual ind = null;
            // Code à mettre ultérieurement ici
            return ind;
    }
```

Son code sera complété à chaque problème particulier.

9.1.4 IHM

Pour découpler l'algorithme génétique de son utilisation (application de bureau, mobile, en ligne de commande, dans un site web...), une interface pour l'IHM (nommée **IIHM**) est définie. Elle devra être implémentée par tous les programmes. Une seule méthode y est présente, celle qui permet d'afficher le meilleur individu à chaque génération.

La méthode doit donc prendre en paramètre l'individu et la génération.

```
public interface IIHM
{
    void PrintBestIndividual(Individual individual, int generation);
}
```

9.1.5 Processus évolutionnaire

La classe principale, **EvolutionaryProcess**, est la dernière. C'est elle qui contrôle tout le processus évolutionnaire.

Son code de base est le suivant :

```
using System.Collections.Generic;
using System.Linq;

public class EvolutionaryProcess
{
}
```

Cette classe possède cinq attributs :

- La population active, qui est une liste d'individus.
- Le numéro de la génération active.
- Une référence vers la classe qui sert d'IHM.
- La meilleure fitness rencontrée jusqu'ici.
- Le nom du problème à résoudre.

Nous rajoutons donc les définitions suivantes :

```
    protected List<Individual> population;
    protected int generationNb = 0;
    protected IIHM program = null;
    protected double bestFitness;
    protected string problem;
```

La première méthode est le constructeur. Celui-ci prend deux paramètres : la chaîne représentant le problème à résoudre, et la référence vers l'IHM. L'environnement des individus est tout d'abord initialisé grâce à un appel à la fabrique, puis la première population (dont la taille est définie dans les paramètres) est créée, là encore via `IndividualFactory`.

```
    public EvolutionaryProcess(IIHM _program, string _problem)
    {
          program = _program;
          problem = _problem;
          IndividualFactory.getInstance().Init(problem);
          population = new List<Individual>();
          for (int i = 0; i < Parameters.individualsNb; i++)
          {
          population.Add(IndividualFactory.getInstance().get
Individualproblem));
          }
    }
```

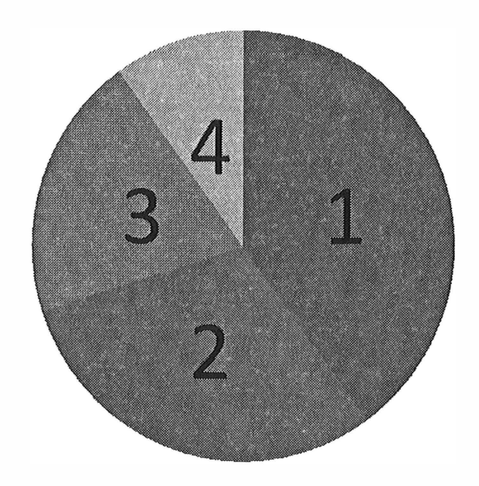
La méthode suivante est celle gérant la survie d'une génération à l'autre. On choisit un simple remplacement : à chaque génération, tous les enfants deviennent les adultes, qui, eux, disparaissent.

Son code est donc très simple :

```
    private void Survival(List<Individual> newGeneration)
    {
          // Remplacement
          population = newGeneration;
    }
```

La prochaine méthode est celle de sélection des individus pour en faire des parents. Celle-ci utilise une roulette biaisée sur le rang. Le premier individu a donc N parts sur la roue, N étant le nombre d'individus. Le deuxième a N-1 part, et ainsi de suite jusqu'au dernier individu qui a une seule part.

Ainsi, à quatre individus, on obtiendrait la roue de loterie suivante (les numéros représentent les individus) :



On choisit donc une part au hasard, et c'est l'individu correspondant qui est renvoyé. La première étape consiste à connaître le nombre total de parts élémentaires. Pour cela, il faut faire la somme :

$$Somme = \ N + (N-1) + (N-2) + \cdots + 1 = \sum_{X=1}^{X=N} X$$

Cette somme vaut :

$$Somme = \frac{N\ (N-1)}{2}$$

On choisit donc une part aléatoirement dans celles présentes. Il faut ensuite savoir à qui elle appartient. Pour cela, on teste si la part tirée au sort appartient au premier individu qui possède N parts. Si ce n'est pas le cas, on ajoute les N-1 parts du deuxième, et on regarde s'il est sélectionné, et ainsi de suite.

On retourne enfin l'individu à l'index voulu (en n'oubliant pas de les trier par fitness croissante d'abord).

Le code est donc le suivant :

```
    private Individual Selection()
    {
            // Roulette biaisée sur le rang
            int totalRanks = Parameters.individualsNb *
 (Parameters.individualsNb + 1) / 2;
            int rand = Parameters.randomGenerator.Next(totalRanks);

            int indIndex = 0;
            int nbParts = Parameters.individualsNb;
            int totalParts = 0;

            while(totalParts < rand) {
                    indIndex++;
                    totalParts += nbParts;
                    nbParts --;
            }

            return population.OrderBy(x =>
x.Fitness).ElementAt(indIndex);
    }
```

La dernière méthode est la méthode `Run()` qui est la boucle principale. Celle-ci doit boucler, jusqu'à atteindre le nombre maximal de générations ou la fitness cible fixés dans les paramètres.

À chaque itération, il faut :

- Lancer l'évaluation de chaque individu.
- Récupérer le meilleur individu et lancer son affichage.
- Créer une nouvelle population (en utilisant l'élitisme pour conserver la meilleure solution jusqu'alors puis la sélection du ou des parents si le crossover s'applique).
- Appliquer la survie des descendants (qui deviennent la population en cours).

Le code est donc le suivant :

```
    public void Run()
    {
            bestFitness = Parameters.minFitness + 1;
            while (generationNb < Parameters.generationsMaxNb &&
bestFitness > Parameters.minFitness)
            {
                    // Évaluation
                    foreach (Individual ind in population)
                    {
                            ind.Evaluate();
                    }

                    // Meilleur individu (stats)
                    Individual bestInd = population.OrderBy(x =>
x.Fitness).FirstOrDefault();
                    program.PrintBestIndividual(bestInd, generationNb);
                    bestFitness = bestInd.Fitness;

                    // Sélection et reproduction
                   List<Individual> newGeneration = new List <Individual>();
                    // Élitisme :
                    newGeneration.Add(bestInd);
                    for (int i = 0; i < Parameters.individualsNb - 1; i++)
                    {
                            // Un ou deux parents ?
                            if (Parameters.randomGenerator.NextDouble() <
Parameters.crossoverRate)
                            {
                                    // Choisir parents
                                    Individual father = Selection();
                                    Individual mother = Selection();

                                    // Reproduction

   newGeneration.Add(IndividualFactory.getInstance().getIndividual
(problem, father, mother));
                            }
                            else
                            {
                                    // Choisir parent
                                    Individual father = Selection();

                                    // Reproduction
   newGeneration.Add(IndividualFactory.getInstance().getIndividual
(problem, father));
                            }
```

```
                }

                // Survie
                Survival(newGeneration);

                generationNb++;
            }
        }
```

Notre algorithme génétique générique est maintenant complet. Il ne reste plus qu'à coder les problèmes à résoudre.

9.2 Utilisation pour le voyageur de commerce

9.2.1 Présentation du problème

Le premier problème est un grand classique en informatique : le problème du voyageur de commerce (en anglais *Travelling Salesman Problem* ou TSP).

On cherche à minimiser le nombre de kilomètres à faire pour un vendeur qui doit passer une seule fois par chaque ville et revenir à son point de départ.

S'il y a cinq villes, on peut choisir n'importe quelle ville de départ, on a donc cinq choix. Il nous reste ensuite quatre villes à visiter pour le deuxième choix, puis trois ensuite et ainsi de suite. Il existe donc 5 * 4 * 3 * 2 * 1 chemins possibles, soit 120. Pour N villes, il existe N * (N-1) * (N-2) * ... * 1, que l'on note N! (factorielle N).

Pour six villes, on passe donc de 120 à 720 enchaînements possibles. À sept villes, il y a 5040 possibilités. Avec dix villes, on arrive à presque 4 millions de possibilités !

Le voyageur de commerce fait partie des problèmes dits "NP complets" : ceux-ci possèdent un nombre de solutions potentielles qui augmente de façon exponentielle, sans moyen mathématique de déterminer la meilleure.

Lorsque le nombre de villes reste faible, on peut encore tester toutes les possibilités. Cependant, très vite, on est bloqués. Il est alors intéressant de passer sur des heuristiques qui ne testent pas tous les chemins, mais seulement les plus intéressants, et c'est le cas des algorithmes génétiques.

Ici, nous travaillerons avec un problème à sept villes françaises, avec les distances suivantes en kilomètres :

Paris						
462	**Lyon**					
772	326	**Marseille**				
379	598	909	**Nantes**			
546	842	555	338	**Bordeaux**		
678	506	407	540	250	**Toulouse**	
215	664	1005	584	792	926	**Lille**

Sur ce problème, l'optimum est de 2579 kilomètres. Il existe 7*2=14 parcours sur les 5040 possibles qui ont cette longueur. En effet, on peut partir de chacune des sept villes pour faire la boucle la plus courte, et on peut la faire dans le sens que l'on souhaite. La probabilité de tomber "au hasard" sur une solution optimale est donc de 0.28 %.

9.2.2 Environnement

Nous commençons donc le code par un ensemble de classes représentant l'environnement de l'individu. Ces classes permettent de définir les gènes et l'évaluation de l'individu.

On commence par définir la structure **`City`** représentant une ville, qui n'est qu'une chaîne de caractères avec le nom de celle-ci. On définit aussi un constructeur pour initialiser la ville et une méthode `ToString` pour l'affichage.

```
using System;
using System.Collections.Generic;

internal struct City
{
    String name;

    public City(string p)
    {
          name = p;
    }

    public override string ToString()
    {
          return name;
    }
}
```

On définit ensuite la classe **TSP**, qui est statique, et qui permet de manipuler les villes. Cette classe représente le problème à résoudre. Elle contient donc une liste de villes (`cities`) et un tableau à double entrée indiquant les distances séparant ces villes (`distances`).

```
using System;
using System.Collections.Generic;

public static class TSP
{
    static List<City> cities;
    static int[][] distances;

    // Méthodes ici

}
```

La première méthode est le constructeur. Celui-ci initialise la liste des villes et la remplit avec les sept villes citées précédemment. Ensuite, il crée le tableau à double entrée et y rentre les différents kilométrages séparant les villes, dans l'ordre.

```
    public static void Init()
    {
          cities = new List<City>()
          {
                new City("Paris"),
```

```
                new City("Lyon"),
                new City("Marseille"),
                new City("Nantes"),
                new City("Bordeaux"),
                new City("Toulouse"),
                new City("Lille")
            };

            distances = new int[cities.Count][];

            distances[0] = new int[] { 0, 462, 772, 379, 546, 678,
215 }; // Paris
            distances[1] = new int[] { 462, 0, 326, 598, 842, 506,
664 }; // Lyon
            distances[2] = new int[] { 772, 326, 0, 909, 555, 407,
1005 }; // Marseille
            distances[3] = new int[] { 379, 598, 909, 0, 338, 540,
584 }; // Nantes
            distances[4] = new int[] { 546, 842, 555, 338, 0, 250,
792 }; // Bordeaux
            distances[5] = new int[] { 678, 506, 407, 540, 250, 0,
926 }; // Toulouse
           distances[6] = new int[] { 215, 664, 1005, 584, 792, 926,
0 }; // Lille
    }
```

On code ensuite une méthode `getDistance` qui permet de savoir combien de kilomètres séparent deux villes passées en paramètres. Pour cela la ville est cherchée dans la liste, et son index est injecté dans le tableau des distances.

```
    internal static int getDistance(City _city1, City _city2)
    {
            return
distances[cities.IndexOf(_city1)][cities.IndexOf(_city2)];
    }
```

Enfin, une dernière méthode est créée : elle renvoie une copie des villes existantes, et se nomme `getCities`. Pour cela, une nouvelle liste de villes est créée à laquelle on ajoute toutes les villes existantes. On ne renvoie pas directement la liste enregistrée en attribut pour être sûr qu'elle ne soit jamais modifiée.

```
    internal static List<City> getCities() {
          List<City> listCities = new List<City>();
          listCities.AddRange(cities);
          return listCities;
    }
```

L'environnement est maintenant complet, il est possible de coder les individus.

9.2.3 Gènes

Les individus sont composés d'une suite de gènes, chacun d'eux représentant une ville à visiter. Nous commençons donc par définir les gènes, avec une classe **TSPGene** qui implémente l'interface IGene.

Cette classe ne contient qu'un attribut, la ville correspondant au gène :

```
using System;

internal class TSPGene : Igene
{
    City city;

    // Méthodes ici
}
```

Les deux premières méthodes sont les constructeurs. Dans les deux cas, il faut indiquer la ville correspondante, soit directement sous la forme d'un objet City (au moment de l'initialisation), soit grâce à un autre gène qu'il faudra copier (pour la reproduction).

Les constructeurs sont donc les suivants :

```
    public TSPGene(City _city)
    {
          city = _city;
    }

    public TSPGene(TSPGene g)
    {
          city = g.city;
    }
```

On ajoute une méthode `getDistance`, qui renvoie la distance entre la ville de ce gène et celle contenue dans un autre gène. Pour cela, on appelle la méthode créée précédemment dans l'environnement `TSP`.

```
internal int getDistance(TSPGene g)
{
        return TSP.getDistance(city, g.city);
}
```

L'interface `IGene` définit une méthode `Mutate()`. Or, dans le cas du problème du voyageur de commerce, la mutation d'un gène seul n'a pas de sens : on ne peut que changer l'ordre des gènes, mais pas leur contenu, étant donné qu'on doit forcément passer une et une seule fois par chaque ville. Cette méthode lève donc une exception si elle est appelée.

```
public void Mutate()
{
      throw new NotImplementedException();
}
```

On termine cette classe par une méthode `ToString`, de manière à pouvoir afficher nos gènes (et donc notre meilleur individu) :

```
public override string ToString()
{
      return city.ToString();
}
```

Les gènes sont maintenant codés. Ce codage est très spécifique au problème à résoudre, mais il reste cependant très rapide.

9.2.4 Individus

Nous pouvons donc maintenant coder nos individus **`TSPIndividual`**, qui héritent de la classe abstraite `Individual`. Le génome et la fitness étant déjà définis, nous n'avons pas d'attributs supplémentaires.

La première méthode est le constructeur par défaut. Celui-ci, appelé lors de l'initialisation, demande la liste des villes à parcourir, puis en choisit une aléatoirement et la transforme en gène, et ainsi de suite, jusqu'à ce que toutes les villes aient été visitées.

```
using System.Collections.Generic;
using System.Linq;

internal class TSPIndividual : Individual
{
    // Constructeur par défaut : initialisation aléatoire
    public TSPIndividual()
    {
          genome = new List<IGene>();
          List<City> cities = TSP.getCities();
          while (cities.Count != 0)
          {
                 int index = Parameters.randomGenerator.
Next(cities.Count);
                genome.Add(new TSPGene(cities.ElementAt(index)));
                 cities.RemoveAt(index);
       }
    }
```

L'opérateur de mutation consiste à changer la place d'un gène : on enlève un gène aléatoirement et on le replace à un index tiré au sort. Cette mutation ne se fait que si on tire un nombre inférieur au taux de mutation.

```
   protected override void Mutate()
   {
          if (Parameters.randomGenerator.NextDouble() <
Parameters.mutationsRate)
          {
                 int index1 =
Parameters.randomGenerator.Next(genome.Count);
                 TSPGene g = (TSPGene)genome.ElementAt(index1);
                 genome.RemoveAt(index1);
                 int index2 =
Parameters.randomGenerator.Next(genome.Count);
                 genome.Insert(index2, g);
          }
   }
```

On peut maintenant coder les deux derniers constructeurs. Le premier est un constructeur utilisé lorsque l'on a un seul parent. Dans ce cas, on reconstruit un génome en faisant une copie des gènes un à un, puis on appelle notre opérateur de mutation.

```
// Constructeur avec un parent (copie + mutations)
public TSPIndividual(TSPIndividual father)
{
	this.genome = new List<IGene>();
	foreach (TSPGene g in father.genome)
	{
		this.genome.Add(new TSPGene(g));
	}
	Mutate();
}
```

Le deuxième est appelé dans le cas d'un crossover. On choisit un point de coupure aléatoirement, et on copie les villes avant ce point depuis le premier parent. On parcourt ensuite le deuxième parent pour ne récupérer que les villes non encore visitées, en conservant leur ordre. Enfin, on appelle l'opérateur de mutation.

```
// Constructeur avec deux parents (crossover et mutations)
public TSPIndividual(TSPIndividual father, TSPIndividual mother)
{
	this.genome = new List<IGene>();
	// Crossover
	int cuttingPoint =
Parameters.randomGenerator.Next(father.genome.Count);
	foreach (TSPGene g in father.genome.Take(cuttingPoint))
	{
		this.genome.Add(new TSPGene(g));
	}
	foreach (TSPGene g in mother.genome)
	{
		if (!genome.Contains(g))
		{
			this.genome.Add(new TSPGene(g));
		}
	}

	// Mutation
	Mutate();
}
```

La dernière méthode de cette classe est l'évaluation d'un individu. Pour cela, on doit parcourir la liste des villes, et demander la distance entre les villes deux à deux. Enfin, on n'oublie pas de rajouter la distance de la dernière à la première ville pour boucler notre parcours.

```
    internal override double Evaluate()
    {
           int totalKm = 0;
           TSPGene oldGene = null;
           foreach (TSPGene g in genome)
           {
                 if (oldGene != null)
                 {
                       totalKm += g.getDistance(oldGene);
                 }
                 oldGene = g;
           }
           totalKm +=
oldGene.getDistance((TSPGene)genome.FirstOrDefault());
           fitness = totalKm;
           return fitness;
    }
```

La classe `TSPIndividual` est terminée. Cependant, nous avons besoin de modifier la fabrique d'individus **`IndividualFactory`** pour qu'elle puisse appeler les bons constructeurs et la bonne initialisation en fonction des besoins. Il faut donc à chaque fois créer un switch, et si le problème vaut "TSP", alors nous appellerons nos différents opérateurs.

La méthode `Init` devient donc :

```
    internal void Init(string type)
    {
           switch (type)
           {
                 case "TSP":
                       TSP.Init();
                       break;
           }
    }
```

On modifie ensuite la méthode permettant d'initialiser un individu :

```
    public Individual getIndividual(String type) {
           Individual ind = null;
           switch (type)
           {
                 case "TSP":
                       ind = new TSPIndividual();
```

```
                        break;
            }
            return ind;
    }
```

On fait de même avec le constructeur à partir d'un parent, puis celui à partir de deux parents, qui appellent les bons constructeurs de notre classe `TSPIndividual`.

```
    public Individual getIndividual(String type, Individual father)
    {
            Individual ind = null;
            switch (type)
            {
                    case "TSP":
                        ind = new TSPIndividual((TSPIndividual)father);
                        break;
            }
            return ind;
    }

    public Individual getIndividual(String type, Individual father,
Individual mother)
    {
            Individual ind = null;
            switch (type)
            {
                    case "TSP":
                        ind = new TSPIndividual((TSPIndividual)father,
(TSPIndividual)mother);
                        break;
            }
            return ind;
    }
```

Notre programme est maintenant entièrement utilisable.

9.2.5 Programme principal

Nous terminons par le programme principal. Celui-ci contient la méthode `main`, et implémente `IIHM` pour pouvoir obtenir et afficher le meilleur individu.

Le squelette de notre classe **Program** est donc le suivant :

```
using System;

class Program : IIHM
{
    static void Main(string[] args)
    {
          Program p = new Program();
          p.Run();
    }

    public void Run()
    {
          // Code principal ICI
          while (true) ;
    }

    public void PrintBestIndividual(Individual individual, int
generation)
    {
          Console.WriteLine(generation + " -> " + individual);
    }
}
```

Le code principal du programme à mettre dans la méthode `Run` est simple. En effet, on commence par mettre les paramètres souhaités. Nous ne voulons pas pouvoir ajouter ou supprimer des gènes, et le crossover n'apporte rien sur un problème aussi petit, du coup les taux correspondants sont à 0. On fixe aussi un taux de mutation à 0.3, indiquant que 30 % des individus subissent un échange dans leur génome. Enfin, on fixe la fitness à atteindre à l'optimum de 2579.

Une fois les paramètres indiqués, on crée un nouveau processus évolutionnaire, en lui précisant que le problème à résoudre est de type "TSP". Et on termine par lancer l'algorithme.

```
    public void Run()
    {
          //Init
          Parameters.crossoverRate = 0.0;
          Parameters.mutationsRate = 0.3;
          Parameters.mutationAddRate = 0.0;
          Parameters.mutationDeleteRate = 0.0;
```

```
            Parameters.minFitness = 2579;
            EvolutionaryProcess geneticAlgoTSP = new
EvolutionaryProcess(this, "TSP");

            // Lancement
            geneticAlgoTSP.Run();

            while (true) ;
    }
```

9.2.6 Résultats

Rien ne permet de s'assurer, avec un algorithme génétique, que toutes les simulations trouvent l'optimum, à moins de laisser l'algorithme tourner indéfiniment. Ici, nous avons limité l'algorithme à 50 générations.

Lors de tests effectués avec les paramètres indiqués, les 50 simulations ont trouvé une solution optimale. Un déroulement classique de l'algorithme est le suivant :

```
0 -> 3313 : Lille - Paris - Nantes - Toulouse - Lyon - Marseille - Bordeaux
1 -> 3313 : Lille - Paris - Nantes - Toulouse - Lyon - Marseille - Bordeaux
2 -> 3313 : Lille - Paris - Nantes - Toulouse - Lyon - Marseille - Bordeaux
3 -> 3313 : Lille - Paris - Nantes - Toulouse - Lyon - Marseille - Bordeaux
4 -> 3313 : Lille - Paris - Nantes - Toulouse - Lyon - Marseille - Bordeaux
5 -> 3067 : Marseille - Bordeaux - Nantes - Lille - Paris - Lyon - Toulouse
6 -> 3067 : Marseille - Bordeaux - Nantes - Lille - Paris - Lyon - Toulouse
7 -> 3067 : Marseille - Bordeaux - Nantes - Lille - Paris - Lyon - Toulouse
8 -> 3067 : Marseille - Bordeaux - Nantes - Lille - Paris - Lyon - Toulouse
9 -> 2991 : Bordeaux - Nantes - Lille - Paris - Marseille - Lyon - Toulouse
10 -> 2991 : Bordeaux - Nantes - Lille - Paris - Marseille - Lyon - Toulouse
11 -> 2991 : Bordeaux - Nantes - Lille - Paris - Marseille - Lyon - Toulouse
12 -> 2582 : Bordeaux - Nantes - Lille - Paris - Lyon - Marseille - Toulouse
13 -> 2582 : Bordeaux - Nantes - Lille - Paris - Lyon - Marseille - Toulouse
14 -> 2582 : Bordeaux - Nantes - Lille - Paris - Lyon - Marseille - Toulouse
15 -> 2579 : Marseille - Lyon - Lille - Paris - Nantes - Bordeaux - Toulouse
```

En moyenne sur les 50 tests, l'algorithme a convergé en 16.1 générations. Au maximum, il y a donc eu 16.1*20 individus et donc solutions potentielles testées, soit 322. Quand on compare aux 5040 solutions possibles (parmi lesquelles se trouvent 14 trajets optimaux), on voit que l'algorithme nous a permis de ne pas tester toutes les possibilités, en étant dirigé via l'évolution vers le but à atteindre.

9.3 Utilisation pour la résolution d'un labyrinthe

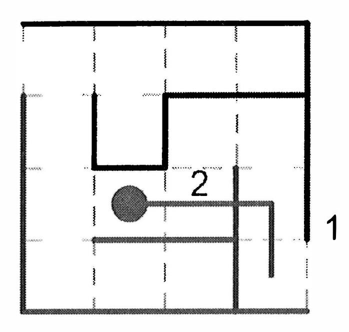
9.3.1 Présentation du problème

Nous voulons trouver la sortie d'un labyrinthe. Pour cela, chaque individu est composé de la suite d'actions à mener pour aller de l'entrée à la sortie.

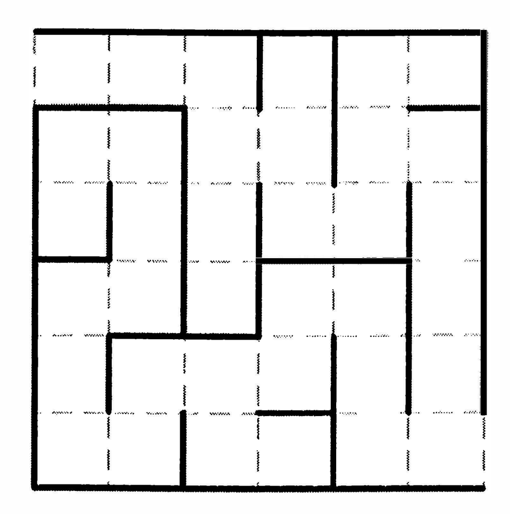
Nous nous arrêtons dès qu'une solution fonctionne, même si elle n'est pas optimale en termes de mouvements (on peut donc faire des allers-retours).

La fitness d'un individu est la distance de Manhattan entre sa dernière position et la sortie. Cette distance est très simple : il s'agit simplement du nombre de cases horizontales ajoutées au nombre de cases verticales entre un individu et la case de sortie.

Dans l'exemple suivant, l'individu (cercle) est à une distance de 3 de la sortie : deux cases horizontales et une case verticale.



En plus de ce premier labyrinthe, un deuxième que voici est proposé :



Remarque

Il existe des algorithmes efficaces pour sortir d'un labyrinthe, l'utilisation d'un algorithme génétique dans une application réelle pour résoudre ce problème ne serait donc certainement pas le meilleur choix.

La difficulté ici réside dans le nombre de chemins possibles, et le fait qu'on ne connait pas à l'avance la taille des chemins. On doit donc gérer des génomes de tailles variables (chaque gène correspondant à un ordre de déplacement).

Les ordres de déplacement sont absolus et s'appliquent jusqu'à ce que celui-ci ne soit plus possible (mur) ou qu'on rencontre un carrefour.

9.3.2 Environnement

Tout comme pour le problème du voyageur de commerce, nous allons commencer par créer l'environnement de nos individus. Ici, nous avons besoin principalement de générer les labyrinthes et de calculer les distances.

Nous commençons par définir une structure **Case** qui correspond à une case de notre labyrinthe, et qui contient les coordonnées de celle-ci, ainsi qu'un constructeur :

```
using System;

struct Case
{
    public int i;
    public int j;

    public Case(int _i, int _j)
    {
           i = _i;
           j = _j;
    }
}
```

Une classe statique **Maze** est définie. Celle-ci contient toutes les "portes" du labyrinthe, c'est-à-dire tous les passages d'une case à une autre. Chacun est donc un couple de cases, que l'on représente par un tuple.

La classe contient donc trois attributs : la liste des portes nommée `paths`, la case d'entrée et la case de sortie.

```
using System;
using System.Collections.Generic;
using System.Linq;

public static class Maze
{
    private static List<Tuple<Case, Case>> paths;
    private static Case entrance;
    private static Case exit;

    // Autres attributs

    // Méthodes
}
```

Il faut aussi définir deux chaînes qui sont nos labyrinthes en "ASCII art". Cela permet de les rentrer plus facilement qu'en créant la liste des portes à la main.

```
public static String Maze1 = "*--*--*--*--*\n" +
                             "E           |\n" +
                             "*  *  *--*--*\n" +
                             "|  |  |     |\n" +
                             "*  *--*  *  *\n" +
                             "|        |  |\n" +
                             "*  *--*--*  *\n" +
                             "|        |  S\n" +
                             "*--*--*--*--*";

public static String Maze2 = "*--*--*--*--*--*--*\n" +
                             "E        |  |     |\n" +
                             "*--*--*  *  *  *--*\n" +
                             "|     |     |     |\n" +
                             "*  *  *  *  *  *  *\n" +
                             "|  |  |  |     |  |\n" +
                             "*--*  *  *--*--*  *\n" +
                             "|     |  |     |  |\n" +
                             "*  *--*--*  *  *  *\n" +
                             "|  |        |  |  |\n" +
                             "*  *  *  *--*  *  *\n" +
                             "|     |     |     S\n" +
                             "*--*--*--*--*--*--*";
```

Enfin, les directions sont définies par une énumération qui peut prendre quatre valeurs :

```
public enum Direction { Top, Bottom, Left, Right };
```

La première méthode initialise les attributs (passages et entrée/sortie) en fonction d'une chaîne passée en paramètre (le dessin du labyrinthe). Pour cela, la chaîne est d'abord découpée sur le caractère '\n' (retour chariot).

Les lignes impaires correspondent aux murs. On peut les séparer sur les caractères '*'. Soit entre deux il y a "--" indiquant la présence d'un mur, soit on a seulement des espaces indiquant l'absence d'un mur. Dans ce dernier cas, on rajoute un passage vertical, de la case du dessus à celle du dessous.

Les lignes impaires correspondent aux couloirs. Il faut alors séparer les caractères trois par trois, représentant une case. En cas de présence d'un caractère '|', on sait alors qu'il y a un mur. En son absence, on peut créer un passage horizontal entre la case et celle d'avant.

On va aussi vérifier la présence de l'entrée et de la sortie lorsqu'on est sur une ligne paire (correspondant aux couloirs). Dans ce cas, on aura un 'E' ou un 'S' dans la ligne.

Le code est donc le suivant :

```
    public static void Init(String s)
    {
           paths = new List<Tuple<Case, Case>>();

           String[] lines = s.Split(new char[] {'\n'},
StringSplitOptions.RemoveEmptyEntries);
           int nbLines = 0;
           foreach (String line in lines)
           {
                  if (nbLines % 2 != 0)
                  {
                         // Ligne impaire, donc contenu d'un couloir
                         int index = line.IndexOf('E');
                         if (index != -1)
                         {
                                if (index == line.Length - 1)
                                {
                                       index--;
                                }
                                entrance = new Case(nbLines / 2,
index / 3);
                         }
                         else
                         {
                                index = line.IndexOf('S');
                                if (index != -1)
                                {
                                       if (index == line.Length-1)
                                       {
                                              index--;
                                       }
                                       exit = new Case(nbLines / 2,
index / 3);
                                }
```

```
                        }
                        for (int column = 0; column < line.Length
/ 3; column++)
                        {
                                String caseStr =
line.Substring(column * 3, 3);
                                if (!caseStr.Contains("|") &&
!caseStr.Contains("E") && !caseStr.Contains("S"))
                                {
                                        paths.Add(new Tuple<Case,
Case>(new Case(nbLines / 2, column - 1), new Case(nbLines / 2,
column)));
                                }
                        }
                }
                else
                {
                        // Ligne paire, donc murs
                        String[] cases = line.Split(new char[] {
'*' }, StringSplitOptions.RemoveEmptyEntries);
                        int column = 0;
                        foreach (String mur in cases) {
                                if (mur.Equals("  "))
                                {
                                        paths.Add(new Tuple<Case,
Case>(new Case(nbLines / 2 - 1, column), new Case(nbLines / 2,
column)));
                                }
                                column++;
                        }
                }
                nbLines++;
        }
}
```

La méthode suivante permet de déterminer s'il est possible d'aller d'une case à une autre. Pour cela, on recherche dans les chemins s'il en existe un allant de la case 1 à la case 2 ou de la case 2 à la case 1. En effet, les passages ne sont enregistrés qu'une fois, bien qu'ils se prennent dans les deux sens.

```
private static bool IsPossible(Case pos1, Case pos2)
{
        return paths.Contains(new Tuple<Case, Case>(pos1,
pos2)) || paths.Contains(new Tuple<Case, Case>(pos2, pos1));
}
```

On écrit aussi une méthode permettant de savoir si une case est un carrefour. Pour cela, on compte simplement le nombre de chemins qui arrivent jusqu'à la case. S'il y en a trois ou plus, alors oui, c'est un carrefour, sinon il s'agit simplement d'un couloir (deux chemins) ou d'un cul-de-sac (un chemin).

```
    private static bool IsJunction(Case pos)
    {
            int nbRoads = paths.Count(x => (x.Item1.Equals(pos) ||
x.Item2.Equals(pos)));
            return nbRoads > 2;
    }
```

La dernière méthode, et la plus complexe, est celle qui permet d'évaluer un individu. Pour cela, on le fait partir de l'entrée qui est sa case de départ.

On applique ensuite les gènes un à un et on change à chaque déplacement la case sur laquelle on est. La direction demandée est gardée jusqu'à ce qu'il ne soit plus possible d'avancer ou qu'on arrive à un carrefour. À ce moment-là, on passe au gène suivant. On s'arrête lorsqu'on arrive sur la case d'arrivée ou lorsqu'il n'y a plus de gènes à appliquer.

On calcule enfin la distance de Manhattan à la sortie, que l'on renvoie.

```
    internal static double Evaluate(MazeIndividual individual)
    {
            Case currentPosition = entrance;

            bool end = false;
            foreach (MazeGene g in individual.Genome)
            {
                  switch (g.direction)
                  {
                        case MazeGene.Direction.Bottom :
                               while (IsPossible(currentPosition,
new Case(currentPosition.i + 1, currentPosition.j)) && !end)
                               {
                                      currentPosition.i++;
                                      end =
IsJunction(currentPosition) || currentPosition.Equals(exit);
                               }
                               end = false;
                               break;
                        case MazeGene.Direction.Top:
```

```
                              while (IsPossible(currentPosition,
new Case(currentPosition.i - 1, currentPosition.j)) && !end)
                              {
                                     currentPosition.i--;
                                     end =
IsJunction(currentPosition) || currentPosition.Equals(exit);
                              }
                              end = false;
                              break;
                       case MazeGene.Direction.Right:
                              while (IsPossible(currentPosition,
new Case(currentPosition.i, currentPosition.j + 1)) && !end)
                              {
                                     currentPosition.j++;
                                     end =
IsJunction(currentPosition) || currentPosition.Equals(exit);
                              }
                              end = false;
                              break;
                       case MazeGene.Direction.Left:
                              while (IsPossible(currentPosition,
new Case(currentPosition.i, currentPosition.j - 1)) && !end)
                              {
                                     currentPosition.j--;
                                     end =
IsJunction(currentPosition) || currentPosition.Equals(exit);
                              }
                              end = false;
                              break;
                }
                if (currentPosition.Equals(exit)) {
                      return 0;
                }
          }

          int distance = Math.Abs(exit.i - currentPosition.i) +
Math.Abs(exit.j - currentPosition.j);
          return distance;
 }
```

L'environnement étant prêt, nous pouvons passer à l'implémentation des individus.

9.3.3 Gènes

Nous commençons par la création des gènes, nommés **MazeGene** et qui implémentent l'interface `IGene`. Un gène contient uniquement une direction à suivre. Deux constructeurs sont ajoutés : un pour créer un gène aléatoirement, et un pour copier un gène donné en paramètre.

```
internal class MazeGene : Igene
{
    public Maze.Direction direction;

    public MazeGene()
    {
        direction =
(Maze.Direction)Parameters.randomGenerator.Next(4);
    }

    public MazeGene(MazeGene g)
    {
        direction = g.direction;
    }
}
```

Nous ajoutons une méthode `ToString`, qui n'affiche que la première lettre de la direction pour simplifier les affichages :

- B pour Bottom (bas).
- T pour Top (haut).
- L pour Left (gauche).
- R pour Right (droite).

```
public override string ToString()
{
    return direction.ToString().Substring(0,1);
}
```

Enfin il faut définir une méthode `Mutate` pour respecter l'interface. Celle-ci se contente de refaire un tirage au sort pour une nouvelle direction. Comme il y a quatre directions possibles, on retombera dans 25 % des cas sur la direction que l'on avait avant la mutation.

```
    public void Mutate()
    {
           direction = (Maze.Direction)
Parameters.randomGenerator.Next(4);
    }
```

9.3.4 Individus

Nous allons maintenant coder les individus dans la classe **`MazeIndividual`** qui hérite de `Individual`, la classe abstraite définie précédemment.

Cette classe ne contient pas d'attribut, la fitness et le génome étant déjà définis dans la classe parente.

Nous commençons par créer un premier constructeur, qui ne prend pas de paramètre, et qui permet donc de créer des individus aléatoirement. Ceux-ci possèdent autant de gènes que défini dans la classe de paramétrage.

Le code de base de la classe est donc le suivant :

```
using System.Collections.Generic;
using System.Linq;

internal class MazeIndividual : Individual
{
    // Constructeur par défaut : initialisation aléatoire
    public MazeIndividual()
    {
           genome = new List<IGene>();
           for (int i = 0; i < Parameters.initialGenesNb; i++)
           {
                  genome.Add(new MazeGene());
           }
    }
}
```

Il nous faut ensuite une méthode permettant de faire muter nos individus. Cette mutation peut avoir trois formes différentes :

– La suppression d'un gène, avec un taux défini par `mutationDeleteRate`. Le gène est choisi aléatoirement.

– L'ajout d'un gène, avec un taux défini par `mutationAddRate`. Le nouveau gène est ajouté à la suite du parcours déjà créé, et la direction est choisie aléatoirement.

– La modification des gènes, avec un taux par gène de `mutationsRate`. On parcourt donc tout le génome et on teste si on doit changer les directions une à une.

Le code de cette méthode est donc le suivant :

```
    protected override void Mutate()
    {
            // Suppression ?
            if (Parameters.randomGenerator.NextDouble() <
Parameters.mutationDeleteRate)
            {
                int geneIndex =
Parameters.randomGenerator.Next(genome.Count);
                genome.RemoveAt(geneIndex);
            }

            // Ajout ?
            if (Parameters.randomGenerator.NextDouble() <
Parameters.mutationAddRate)
            {
                genome.Add(new MazeGene());
            }

            // Remplacement gène par gène ?
            foreach (MazeGene g in genome)
            {
                if (Parameters.randomGenerator.NextDouble() <
Parameters.mutationsRate)
                {
                    g.Mutate();
                }
            }
    }
```

On crée ensuite un deuxième constructeur qui ne prend qu'un parent. Dans ce cas, on copie les gènes un à un, puis on appelle la méthode de mutation :

```
// Constructeur avec un parent (copie + mutations)
public MazeIndividual(MazeIndividual father)
{
        this.genome = new List<IGene>();
        foreach (MazeGene g in father.genome)
        {
                this.genome.Add(new MazeGene(g));
        }
        Mutate();
}
```

Le dernier constructeur prend deux parents en paramètres. On choisit tout d'abord un point de crossover aléatoirement. Tous les gènes avant ce point sont copiés depuis le premier parent, puis c'est le cas des gènes situés après ce point dans le deuxième parent (s'il reste des gènes). Pour cela, on utilise avantageusement les méthodes `Take` et `Skip` qui permettent de prendre les n premiers éléments d'une collection ou au contraire de ne prendre que les suivants.

```
// Constructeur avec deux parents (crossover et mutations)
public MazeIndividual(MazeIndividual father, MazeIndividual
mother)
{
        this.genome = new List<IGene>();
        // Crossover
        int cuttingPoint =
Parameters.randomGenerator.Next(father.genome.Count);
        foreach (MazeGene g in
father.genome.Take(cuttingPoint))
        {
                this.genome.Add(new MazeGene(g));
        }
        foreach (MazeGene g in
mother.genome.Skip(cuttingPoint))
        {
                this.genome.Add(new MazeGene(g));
        }
        // Mutation
        Mutate();
}
```

La dernière méthode de cette classe est l'évaluation. On se contente d'appeler la classe `Maze`, qui a une méthode spécifique pour évaluer les individus.

```
    internal override double Evaluate()
    {
           fitness = Maze.Evaluate(this);
           return fitness;
    }
```

Une fois les individus codés, il faut modifier la fabrique d'individus, **`IndividualFactory`**, pour ajouter à chaque méthode un nouveau cas, si le problème donné s'appelle "Maze". Les lignes rajoutées sont en gras dans le code suivant :

```
    public Individual getIndividual(String type) {
           Individual ind = null;
           switch (type)
           {
                 case "Maze" :
                        ind = new MazeIndividual();
                        break;
                 case "TSP":
                       ind = new TSPIndividual();
                       break;
           }
           return ind;
    }

    public Individual getIndividual(String type, Individual
 father)
    {
           Individual ind = null;
           switch (type)
           {
                 case "Maze" :
                        ind = new MazeIndividual((MazeIndividual)
 father);
                        break;
                 case "TSP":
                       ind = new
 TSPIndividual((TSPIndividual)father);
                       break;
           }
           return ind;
```

```
    }

    public Individual getIndividual(String type, Individual father, 
Individual mother)
    {
        Individual ind = null;
        switch (type)
        {
                case "Maze" :
                        ind = new 
MazeIndividual((MazeIndividual)father, (MazeIndividual) mother);
                        break;
                case "TSP":
                        ind = new 
TSPIndividual((TSPIndividual)father, (TSPIndividual)mother);
                        break;
            }
            return ind;
    }

    internal void Init(string type)
    {
            switch (type)
            {
                case "Maze":
                        Maze.Init(Maze.Maze2);
                        break;
                case "TSP":
                        TSP.Init();
                        break;
            }
    }
```

On peut remarquer que le labyrinthe choisi est ici le deuxième, plus complexe que le premier. Il suffit de changer le labyrinthe de la fonction `Init` pour changer le problème.

Remarque

On pourrait rajouter un paramètre à la méthode `Init` pour déterminer la chaîne à utiliser, ou faire une méthode spécifique, mais nous conservons le choix "en dur" pour simplifier le code, et nous focaliser sur l'algorithme génétique.

9.3.5 Programme principal

On termine maintenant par le programme principal **Program**. Il reprend la même structure que celui du problème du voyageur de commerce. La seule différence se situe dans la méthode Run.

On y trouve ainsi des paramètres plus adaptés à ce problème :

- Un taux de crossover de 60 %, soit 0.6.
- Des mutations au taux de 0.1 (un gène sur 10 en moyenne), l'ajout d'un gène dans 20 % des cas (0.2) et la suppression dans 10 % (de cette façon, on a tendance à rallonger les chemins plutôt qu'à les raccourcir).
- La fitness minimale visée est nulle, c'est-à-dire que l'on arrive sur la case de sortie.

Le programme est ensuite lancé via la méthode Run du processus.

```
using GeneticAlgorithm;
using System;

class Program : IIHM
{
    static void Main(string[] args)
    {
           Program p = new Program();
           p.Run();
    }

    public void Run()
    {
           // Init
           Parameters.crossoverRate = 0.6;
           Parameters.mutationsRate = 0.1;
           Parameters.mutationAddRate = 0.2;
           Parameters.mutationDeleteRate = 0.1;
           Parameters.minFitness = 0;
           EvolutionaryProcess geneticAlgoMaze = new
EvolutionaryProcess(this, "Maze");

           // Lancement
           geneticAlgoMaze.Run();

           while (true) ;
```

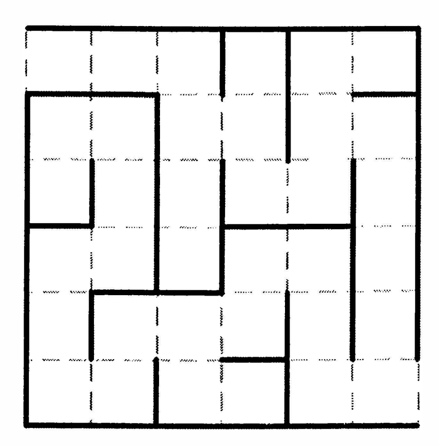
```
    }

    public void PrintBestIndividual(Individual individual, int
generation)
    {
            Console.WriteLine(generation + " -> " + individual);
    }
}
```

Notre programme est terminé !

9.3.6 Résultats

Tout d'abord, rappelons le dessin du labyrinthe utilisé :

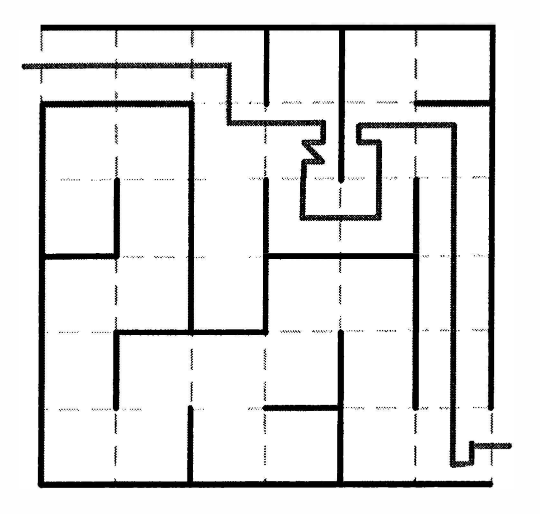


Voici un cas typique de solution obtenue :

```
0 -> 5 : R - B - L - R - B - B - L - B - L - L
1 -> 4 : T - B - R - T - T - T - B - R - B - R
2 -> 4 : T - B - R - T - T - T - B - R - B - R
3 -> 4 : T - B - R - T - T - T - B - R - B - R
4 -> 4 : T - B - R - T - T - T - B - R - B - R
5 -> 4 : T - B - R - T - T - T - B - R - B - R
6 -> 4 : T - B - R - T - T - T - B - R - B - R
7 -> 4 : T - B - R - T - T - T - B - R - B - R
8 -> 4 : T - B - R - T - T - T - B - R - B - R
9 -> 4 : T - B - R - T - T - T - B - R - B - R
10 -> 4 : T - B - R - T - T - T - B - R - B - R
11 -> 0 : R - B - R - R - R - B - R - T - L - R - B - B
```

On peut voir qu'à la première génération, l'algorithme s'arrête à 5 cases de la sortie. Il trouve ensuite des solutions s'arrêtant à 4 cases de la sortie, puis la sortie grâce à l'ajout de gènes. En effet, il est très difficile de résoudre le deuxième labyrinthe en 10 gènes, surtout que l'algorithme fait parfois des allers-retours.

On obtient donc le chemin suivant, et on peut observer qu'il bute contre les murs à trois reprises (notées par les petits décrochements dans le parcours) :



Avec 20 individus sur 50 générations, et les paramètres choisis, on ne tombe cependant pas toujours sur une solution. En effet, il arrive que l'algorithme reste bloqué dans des optimums locaux (sur 50 lancements que nous avons effectués, c'est arrivé quatre fois).

C'est pourquoi lorsqu'on utilise un algorithme génétique, on a tendance à le lancer sur de nombreuses générations et avec beaucoup d'individus, voire à relancer l'algorithme de nombreuses fois pour déterminer plusieurs réponses : rien ne peut garantir la convergence vers la solution dans un temps fini.

10. Coévolution

La **coévolution** est un phénomène biologique formalisé en 1964, suite à une étude de deux biologistes sur les liens entre des plantes et certains papillons. En effet, ils ont montré que les deux espèces avaient évolué conjointement : les papillons mangaient la plante, qui a développé des mécanismes pour se défendre (poison ou protections physiques). Les papillons ont alors développées des moyens de résister à ces poisons ou à ces défenses.

Il s'agit donc d'une course sans fin entre les deux espèces, essayant d'avoir un temps d'avance sur l'autre, pour assurer sa propre survie, et ne pouvant arrêter d'évoluer. Les deux espèces en compétition se sont développées et ont évolué plus vite que si chacune avait évolué de manière distincte.

Remarque

C'est cette coévolution que l'on observe dans tous les systèmes "proies - prédateurs". Ainsi, il existe une course évolutive entre les hackers, qui essaient de percer les sécurités des systèmes informatiques, et les responsables sécurité, qui essaient de repousser et d'empêcher les attaques. Chaque camp doit évoluer en permanence, et rapidement, pour essayer de conserver un avantage sur l'autre camp, et contrer toute nouvelle menace/défense mise en place.

Cette pression évolutive peut être utilisée à notre avantage dans un algorithme génétique. On peut ainsi ne pas faire évoluer une population, mais deux ou plus, les populations entrant en compétition (ou dans de rares cas en coopération). On pourrait ainsi faire évoluer des labyrinthes et des algorithmes de parcours de ces labyrinthes en parallèle.

En robotique, on peut ainsi utiliser un algorithme génétique pour appendre la marche, en commençant par des environnements plats (nos "proies" simples pour commencer). Une fois le robot capable de se déplacer sur des surfaces simples, on pourrait alors faire évoluer le sol, pour rajouter des obstacles, des marches, des trous, etc., et ainsi améliorer le processus de marche.

Cette coévolution demande la modification de la fonction d'évaluation : il faut alors qu'un individu d'une espèce soit évalué par son interaction avec les individus de l'autre espèce. Dans l'exemple de notre robot, les meilleurs sols seraient ceux mettant en échec le robot et les meilleurs robots ceux capables de se déplacer sur le plus de terrains. Au fur et à mesure des générations, la qualité des deux populations augmenterait.

Dans la pratique, cela est rarement mis en place, mais les résultats sont alors bien meilleurs, l'algorithme commençant par résoudre des problèmes simples, puis de plus en plus complexes au fur et à mesure que les problèmes et les solutions évoluent.

11. Synthèse

Les algorithmes génétiques (ou plus généralement les algorithmes évolutionnaires) sont inspirés des différentes recherches faites en biologie sur l'évolution.

On a alors une population d'individus, chacune composée d'un génome, qui est une liste de gènes. Ces individus sont évalués par rapport à la qualité de la solution à un problème donné qu'ils représentent (ce qu'on appelle son phénotype).

Les meilleurs individus sont sélectionnés pour être des reproducteurs. De nouvelles solutions sont alors créées, à partir d'un ou plusieurs parents. Dans le cas où plusieurs parents interviennent, on réalise un crossover, c'est-à-dire un croisement entre les informations génétiques des différents parents.

Les génomes des descendants subissent ensuite des mutations aléatoires, représentant les erreurs de copie qui ont lieu lors de la reproduction. Chaque descendant, bien que proche de ses parents, en est donc potentiellement différent.

Cette nouvelle génération doit ensuite survivre, pour faire partie de la population de la génération suivante.

Les meilleurs individus servant de base aux descendants, la qualité globale de la population s'améliore à chaque génération. Les mutations, elles, permettent de découvrir de nouvelles solutions, potentiellement plus intéressantes que celles testées jusqu'à présent. Au cours du temps, les meilleurs individus répondent de mieux en mieux au problème posé, jusqu'à trouver une solution acceptable au problème, voire la solution optimale.

Pour que ces algorithmes fonctionnent correctement, il faut cependant choisir les différents opérateurs de manière adaptée, ainsi que la représentation des individus. En effet, de mauvais choix peuvent rendre l'ensemble inopérant.

Bien conçus, ils permettent de résoudre des problèmes que l'on ne peut résoudre autrement, à cause du nombre de solutions à tester ou par l'absence de résolution théorique. Ils sont souvent rapides et efficaces, et ne nécessitent qu'une implémentation simple.

Chapitre 5
Métaheuristiques d'optimisation

1. Présentation du chapitre

Ce chapitre présente différentes techniques (ou métaheuristiques) de recherche de minimums locaux. Par exemple, on peut vouloir minimiser un coût de production, ou la quantité de matière nécessaire à une pièce, le tout en respectant de nombreuses contraintes. Ces problèmes sont très courants dans la vie de tous les jours, et pourtant ils sont difficiles à résoudre par un ordinateur (et encore plus par un humain) car le nombre de solutions potentielles est très important.

La première partie de ce chapitre présente donc plus en détail ce problème et les contraintes associées, ainsi que des exemples.

Les parties suivantes présentent les principaux algorithmes : algorithme glouton, descente de gradient, recherche tabou, recuit simulé et optimisation par essaims particulaires.

Les principaux domaines d'application de ces techniques sont ensuite présentés.

Les différents algorithmes sont implémentés dans la dernière partie, en C#. Le code correspondant est proposé en téléchargement.

Enfin, une petite synthèse clôt ce chapitre.

2. Optimisation et minimums

Les problèmes d'optimisation et de recherche de minimums sont courants, et des résolutions exactes sont compliquées, voire impossibles. L'intelligence artificielle a donc mis au point des algorithmes spécifiquement pour ces problèmes.

2.1 Exemples

Les ingénieurs ont à résoudre de nombreux problèmes d'**optimisation**, comme minimiser le coût d'un objet, tout en lui conservant certaines propriétés de résistance, ou optimiser la formule d'un métal pour le rendre plus résistant.

Dans la vie courante, il y a aussi des problèmes de ce type. Payer en utilisant le moins de pièces possible (ou au contraire en essayant de passer un maximum de petites pièces) en est un exemple classique. Pour ceux qui ont des tickets restaurant, commander dans un restaurant ou acheter dans un commerce, assez pour couvrir le prix du ticket (car la monnaie n'est pas rendue dessus) mais en dépassant le moins possible en est un autre.

Charger une voiture, ranger un entrepôt, modifier une composition, déterminer un design, créer un circuit imprimé, limiter les coûts d'emballage... sont autant de problèmes d'optimisation.

2.2 Le problème du sac à dos

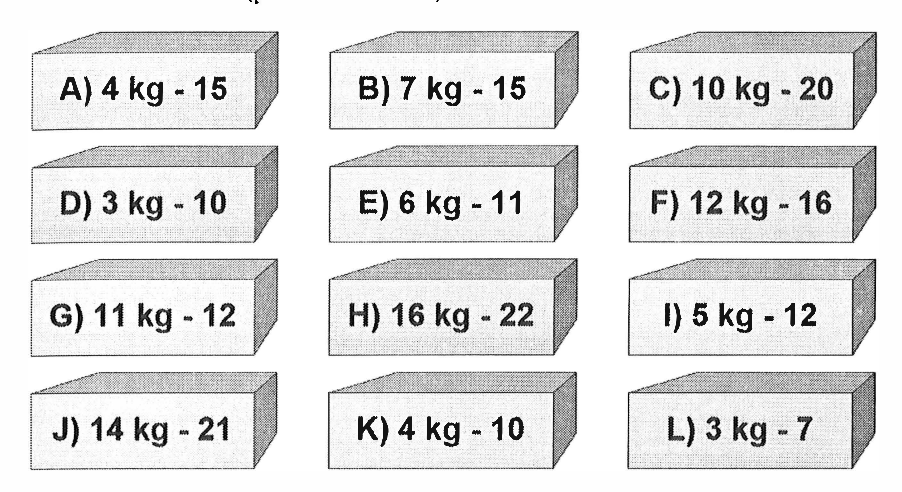
Le **problème du sac à dos** (ou Knapsack Problem en anglais, abrégé en KP) est simple à comprendre mais très difficile à résoudre.

Un sac à dos a une contenance maximale (sinon il risquerait de casser). Plusieurs objets sont disponibles, chacun ayant un poids et une valeur. Le but est de maximiser la valeur des objets chargés.

Bien évidemment, l'ensemble des objets ne peut être chargé (car c'est trop lourd). Il faut donc choisir intelligemment.

Tester toutes les possibilités devient très vite impossible quand le nombre d'objets augmente. En effet, il faut tester toutes les combinaisons de 1 objet, de 2, 3... jusqu'à tous les prendre, et éliminer les solutions impossibles car trop lourdes puis choisir la meilleure.

Imaginons un sac à dos ayant une contenance de 20 kg. Les objets disponibles sont les suivants (poids et valeur) :



Il est par exemple possible de charger les objets A et H, ce qui fait 4+16 = 20 kg et un total de 15+22 = 37 de valeur. Ce n'est cependant pas le meilleur choix. Une autre solution (qui n'est pas optimale non plus, mais meilleure) consiste à charger C, I et K. On a alors un poids de 10+5+4 = 19 kg et une valeur totale de 20+12+10 = 42.

Le chargement optimal est A, D, I, K et L. On a alors un poids de 4+3+5+4+3 = 19 kg, et une valeur totale de 15+10+12+10+7 = 54.

2.3 Formulation des problèmes

Tous les problèmes d'optimisation peuvent s'exprimer de la même façon : il existe une fonction f qui associe une valeur à une solution x, notée f(x). On connait un moyen rapide de la calculer.

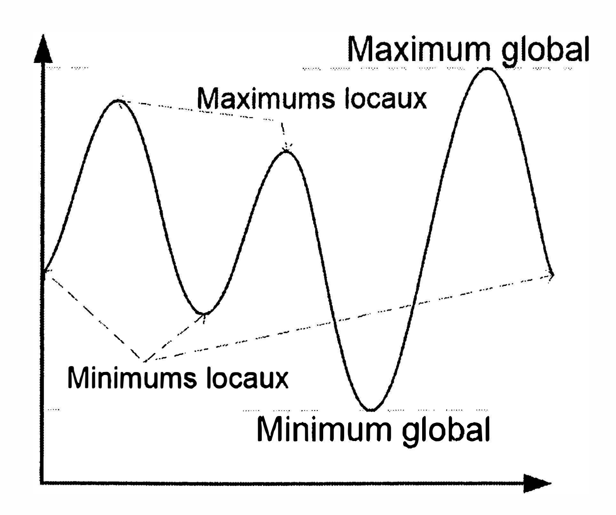
Il y a cependant des contraintes sur x, et les solutions doivent donc appartenir à l'ensemble X des solutions acceptables. Dans le cas du sac à dos, c'est l'ensemble des charges de façon à ce que la somme des poids soit inférieure ou égale à 20 kg.

L'optimisation consiste à trouver x, de façon à minimiser ou à maximiser f(x).

Remarque

Dans la pratique, on ne s'intéresse qu'aux minimisations. En effet, chercher à maximiser f(x) revient à minimiser -f(x). Les algorithmes seront donc uniquement présentés sur des minimisations.

Les maximums et minimums d'une fonction sont appelés **optimums** (ou optima). Les optimums s'appliquant à la fonction sur tout son ensemble de définition sont appelés **optimums globaux**. Ceux n'étant des optimums que par rapport à un voisinage sont appelés **optimums locaux**.



2.4 Résolution mathématique

La première solution qui vient à l'esprit serait d'étudier la fonction f mathématiquement et de trouver le minimum global de la fonction. Cela est possible sur des fonctions ayant une formulation mathématique simple, et c'est d'ailleurs au programme de certaines filières au lycée. Par exemple, la fonction $f(x) = 3 + 1/x$, avec x appartenant à l'ensemble [1,2] a un minimum en 2 (elle vaut alors 3.5).

Cependant, dans la réalité, les fonctions peuvent être soit trop complexes à écrire, soit difficiles à étudier pour trouver ce minimum mathématique. C'est par exemple le cas du sac à dos.

En effet, chaque solution peut être décrite comme un vecteur de dimension N (N étant le nombre d'objets possibles), avec chaque valeur valant 0 ou 1 (respectivement pour un objet ignoré ou chargé). Le poids d'un objet est noté p et sa valeur v.

Dans notre problème, on a 12 objets et donc un vecteur de dimension 12. La deuxième solution (C, I et K) s'exprime alors :

x = (0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0)

Les solutions acceptables sont celles respectant un poids maximum de 20 kg, ce qui mathématiquement s'exprime sous la forme :

$$x, \sum_{i=1}^{N} p(x_i) \leq 20$$

Remarque

Ceci se lit l'ensemble des x, tels que la somme des poids des composants soit inférieure à 20 kg.

La fonction à maximiser est :

$$f(x) = \sum_{i=1}^{n} x_i * v(x_i)$$

Remarque

On cherche donc à maximiser la somme des valeurs des objets pris (qui valent donc 1).

La dérivée de cette fonction n'est pas exprimable, et son étude n'est donc pas possible avec les techniques mathématiques classiques. Trouver un optimum mathématique n'est donc pas la solution.

2.5 Recherche exhaustive

La deuxième solution, après la résolution mathématique, est la **recherche exhaustive**. En testant toutes les possibilités, la solution optimale est forcément trouvée.

Cependant, cette recherche est elle aussi bien souvent trop longue. Dans le problème du sac à dos, il existe trop de solutions possibles, et leur nombre augmente de manière exponentielle avec le nombre d'objets possibles.

Pour les fonctions dont l'espace de recherche est l'ensemble des nombres réels, le problème est encore plus complexe : en effet, le nombre de valeurs entre deux autres valeurs est toujours infini. Tester toutes les possibilités est donc impossible.

Par exemple, si on cherche à minimiser f(x) avec x entre 1 et 2, il existe une infinité de x potentiels. On peut prendre x = 0.5 et x = 0.6. On peut aussi choisir 0.55, 0.578, 0.5896...

La recherche exhaustive est donc, dans le meilleur des cas, trop longue et dans le pire des cas complètement impossible.

2.6 Métaheuristiques

Ne pouvant résoudre de manière déterministe ces problèmes complexes d'optimisation, il est nécessaire d'utiliser d'autres méthodes.

Il existe une famille de méthodes appelées **métaheuristiques**. Celles-ci possèdent plusieurs caractéristiques :

- Elles sont génériques, et peuvent s'adapter à un grand nombre de problèmes.
- Elles sont itératives, c'est-à-dire qu'elles cherchent à améliorer les résultats au fur et à mesure.
- Elles sont souvent stochastiques, c'est-à-dire qu'elles utilisent une part plus ou moins importante de hasard.
- Elles ne garantissent pas de trouver l'optimum global (sauf si on les laisse tourner pendant une durée infinie) mais au moins un optimum local d'assez bonne qualité.

Différentes métaheuristiques simples existent pour l'optimisation et la recherche d'optimums. Ce qui les différencie est la façon dont les solutions vont changer et s'améliorer au cours du temps.

3. Algorithmes gloutons

Les **algorithmes gloutons** sont les plus simples. Ils ne construisent qu'une seule solution, mais de manière itérative. Ainsi, à chaque pas de temps, on rajoute un élément, le plus prometteur.

Cet algorithme est à adapter à chaque problème. Seul le principe général reste le même.

Ainsi, dans le cas du sac à dos, on va rajouter au fur et à mesure les objets les plus intéressants jusqu'à atteindre la capacité du sac.

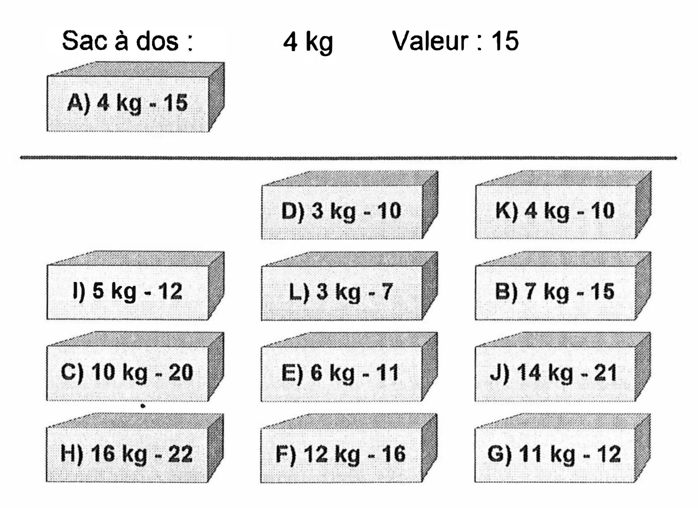
Pour cela, on commence par calculer la valeur par kilo de chaque objet :

A) 4kg - 15 : 3.75	B) 7 kg - 15 : 2.14	C) 10 kg - 20 : 2	D) 3 kg - 10 : 3.33
E) 6 kg - 11 : 1.83	F) 12 kg - 16 : 1.33	G) 11 kg - 12 : 1.09	H) 16 kg - 22 : 1.38
I) 5 kg - 12 : 2.4	J) 14 kg - 21 : 1.5	K) 4 kg - 10 : 2.5	L) 3 kg - 7 : 2.33

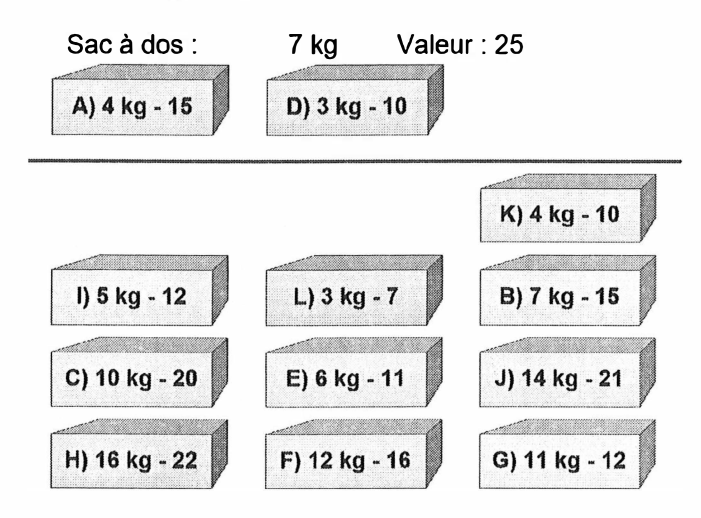
On trie ensuite chaque objet du plus intéressant (la valeur par kilo la plus haute) au moins intéressant. L'ordre obtenu est le suivant :

A - D - K - I - L - B - C - E - J - H - F - G

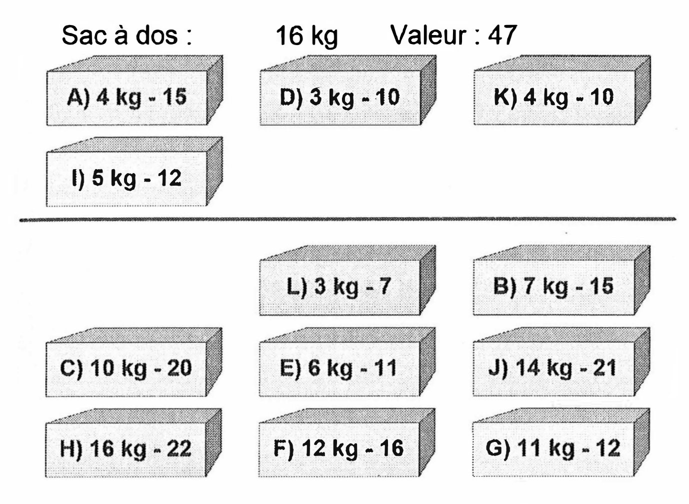
On part d'un sac à dos vide. On ajoute le premier élément d'après le tri, donc l'objet A. Le sac à dos contient alors 4 kg et a une valeur totale de 15.



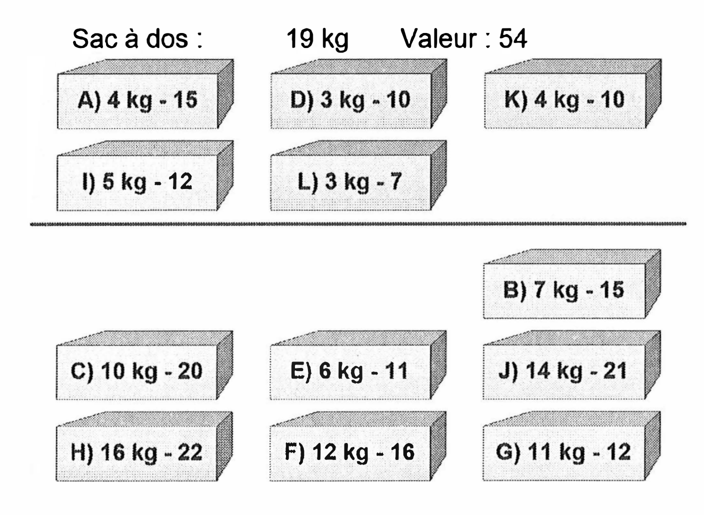
On ajoute ensuite le premier élément de la liste triée restante. L'objet D ne pèse que 3 kg, on peut donc le mettre dans le sac.



Celui-ci contient alors 7 kg et a une valeur de 25. L'élément suivant est K. Après son ajout, le sac à dos contient 11 kg et a une valeur de 35. Le quatrième élément est I.



On a alors 16 kg et une valeur totale de 47. Le prochain élément est L. Le sac à dos contient maintenant 19 kg et a une valeur de 54.



Les éléments suivants de la liste ont tous un poids trop important pour entrer (il ne reste qu'un kilogramme autorisé). L'algorithme s'arrête alors : la solution proposée consiste à mettre dans le sac à dos les objets A, D, I, K et L, pour une valeur de 54. Il s'agit d'ailleurs de l'optimum global sur cet exemple.

4. Descente de gradient

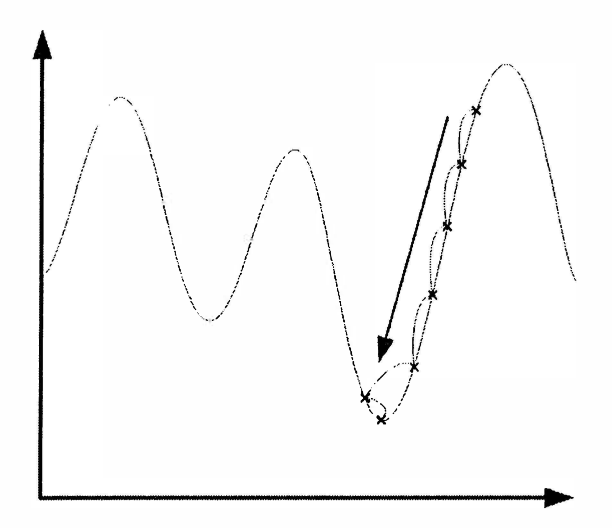
La **descente de gradient** est une métaheuristique incrémentale. À partir d'une première solution, choisie aléatoirement ou donnée comme base (par exemple la meilleure solution connue des experts), l'algorithme va chercher une optimisation en ne modifiant la solution que d'une unité.

Lorsque le problème est une fonction mathématique, on calcule la dérivée au point représentant la solution actuelle, et on suit la direction de la dérivée la plus forte négativement.

Remarque

La dérivée d'une fonction représente sa pente : si elle est positive, alors la courbe est croissante, sinon elle est décroissante. De plus, plus la dérivée est importante et plus la pente est forte.

Sur le schéma suivant, les différentes solutions obtenues itérativement sont indiquées : on part de la solution la plus haute, et on va aller dans le sens du gradient, jusqu'à atteindre le minimum.



Intuitivement, c'est l'algorithme utilisé par un randonneur en forêt : si celui-ci cherche à atteindre le sommet d'une montagne, mais qu'il ne peut savoir où celui-ci se trouve, alors il regarde autour de lui dans quelle direction le chemin monte le plus et le suit. À force de monter, il va forcément se retrouver en haut du massif sur lequel il est. La démarche est la même s'il cherche à atteindre la vallée en suivant les chemins qui descendent.

Généralement, la dérivée mathématique n'est pas accessible. Il n'est donc pas possible de directement suivre celle-ci.

À la place, on va calculer les solutions voisines, à une distance d'une unité (à définir). Chaque solution est ensuite évaluée. Si une meilleure solution est trouvée, alors on repart de cette solution pour une nouvelle itération. En l'absence d'amélioration, on s'arrête.

Comme pour l'algorithme glouton, le choix de l'unité de modification dépend du problème à résoudre. Il n'est donc pas possible de créer un algorithme vraiment très générique.

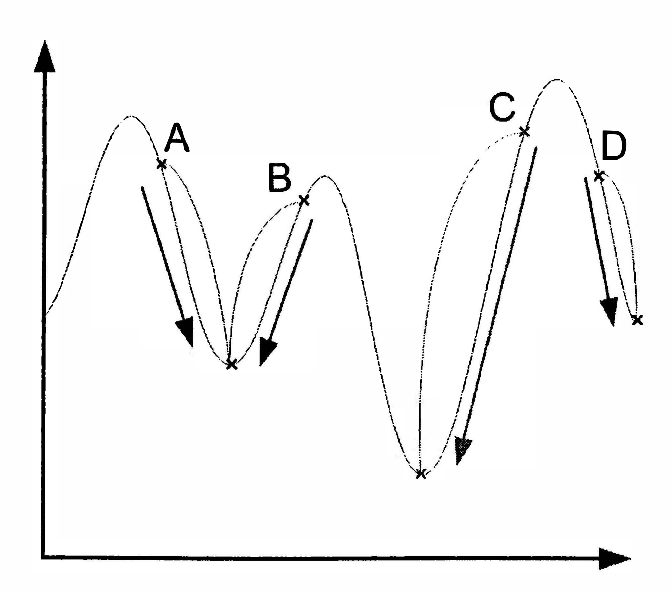
Dans le cas du problème du sac à dos, on peut partir d'une solution aléatoire. Il faut alors tester toutes les variations, en ajoutant un objet non encore sélectionné ou en supprimant un objet mis dans le sac. Cependant enlever un objet va forcément baisser la valeur du sac à dos, c'est pourquoi on préfère échanger des objets (on en enlève un pour en ajouter d'autres à la place).

Si la solution est acceptable (car elle respecte le poids maximum), alors on l'évalue. Seule la meilleure solution parmi toutes les variations est conservée.

La descente de gradient a cependant quelques défauts :

- C'est un algorithme assez lent, car il doit rechercher toutes les solutions voisines et les évaluer.
- L'algorithme ne trouve qu'un optimum local. En effet, seule la vallée sur laquelle se trouve la solution de départ est étudiée, et donc plus le problème a d'optimums locaux et plus il est difficile de trouver l'optimum global.
- Même dans le cas où la solution d'origine se trouve dans le bon voisinage, si un optimum local a une dérivée plus forte que l'optimum global, il va attirer l'algorithme.

Voici par exemple plusieurs solutions initiales A à D. Seule la position de départ C permet de trouver l'optimum global. Les solutions A, B et D ne permettent de trouver qu'un optimum local.



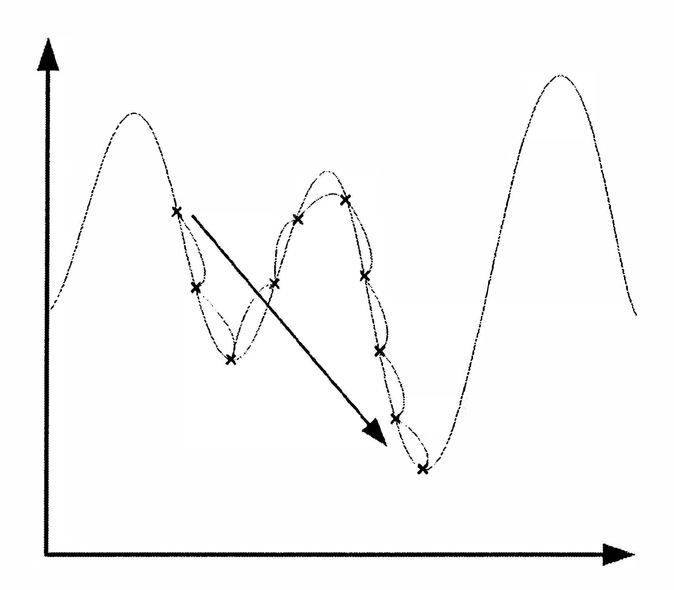
Pour essayer de dépasser ces différents problèmes, on utilise souvent plusieurs initialisations (et donc solutions de départ) pour augmenter le nombre d'optimums découverts et ainsi augmenter les chances de trouver l'optimum global.

5. Recherche tabou

La "**recherche tabou**" est une amélioration de la recherche par descente de gradient. En effet, cette dernière reste bloquée dans le premier optimum rencontré.

Dans le cas de la recherche tabou, à chaque itération, on se déplace vers le meilleur voisin même s'il est moins bon que la solution actuelle. De plus, on retient la liste des positions déjà visitées, qui ne sont plus sélectionnables (d'où le nom, les anciennes solutions deviennent taboues).

De cette façon, l'algorithme se "promène" dans l'espace de solution et ne s'arrête pas au premier optimum découvert. On s'arrête lorsque tous les voisins ont été visités, au bout d'un nombre d'itérations maximal décidé ou lorsqu'aucune amélioration suffisante n'est détectée en x coups.



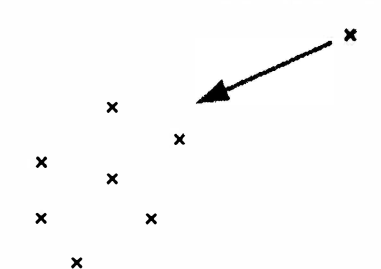
La principale difficulté de cette recherche est le choix de la longueur de la liste de positions taboues. En effet, si cette liste est trop courte, on risque de boucler autour des mêmes positions. Au contraire, une liste trop longue peut empêcher de tester d'autres chemins partant d'une même solution potentielle. Il n'existe cependant aucun moyen de connaître la longueur de la liste idéale, elle doit être choisie de manière purement empirique.

Cette liste est souvent implémentée sous la forme d'une liste (FIFO pour *First In First Out*). De cette façon, une fois que la liste a atteint la taille maximale choisie, les positions enregistrées les plus anciennes sortent des positions taboues.

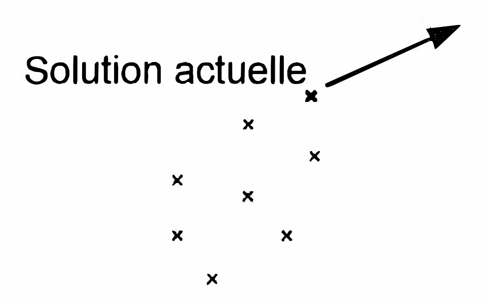
Le choix du voisinage reste identique à celui utilisé par la descente de gradient, et l'application au problème du sac à dos reste donc identique (à savoir ajouter un objet ou faire un échange).

Deux autres processus peuvent être intégrés à la recherche tabou : l'intensification et la diversification.

L'**intensification** consiste à favoriser certaines zones de l'espace qui semblent plus prometteuses. En effet, on enregistre toutes les solutions optimales (locales ou globales) trouvées jusqu'alors. On essaie de tester en priorité les solutions proches de celles-ci, ou possédant les mêmes caractéristiques. Le déplacement est donc biaisé pour se rapprocher de ces solutions :



À l'inverse, la **diversification** a pour but de favoriser la découverte de nouvelles zones de l'espace de recherche. Ainsi, on stocke les positions déjà testées jusqu'à présent et on favorise les solutions différentes. De cette façon, de nouvelles solutions optimales peuvent être découvertes. Le déplacement est alors biaisé pour s'éloigner des anciens optimums :



Ces deux processus doivent être adaptés à chaque problème. S'ils ne sont pas équilibrés, on risque de ne faire que de l'intensification (et donc toujours rester au même endroit) ou au contraire que de la diversification (et donc passer à côté d'optimums proches de ceux déjà découverts).

Là encore, cette adaptation reste empirique.

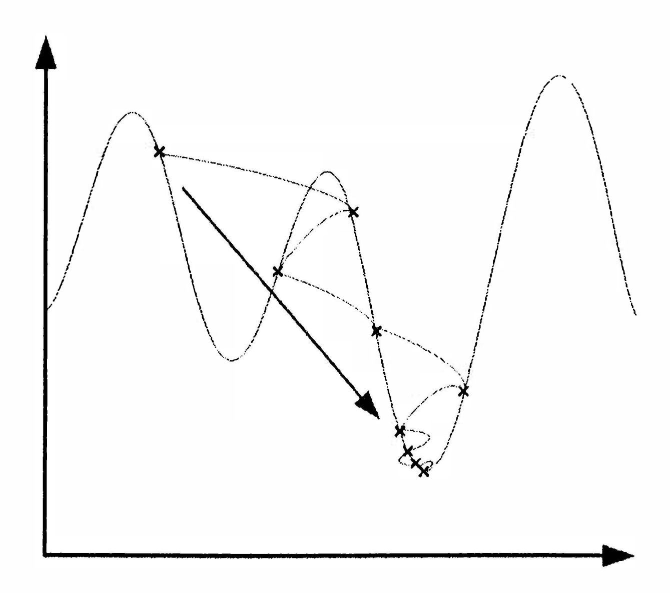
6. Recuit simulé

Le **recuit simulé** améliore la descente de gradient et s'inspire du recuit utilisé en métallurgie. En effet, lorsqu'on forge ou coule des métaux, ceux-ci subissent des contraintes importantes. C'est le cas des lames d'épées par exemple.

Pour augmenter la dureté de la lame, on la réchauffe (d'où le nom de recuit). De cette façon, les atomes peuvent se recristalliser sous des structures plus résistantes, et les contraintes mécaniques et thermiques sont diminuées. Les lames de bonne qualité subissent ainsi plusieurs cycles de chauffe et de mise en forme.

En informatique, on va utiliser ce principe pour améliorer les solutions et sortir des optimums locaux. On va donc fixer une **température** numérique, qui va baisser au cours du temps. Plus cette température est importante et plus les sauts dans l'espace de recherche peuvent être grands. De même, on accepte, contrairement à la descente de gradient, d'aller sur des solutions moins optimales que la solution actuelle.

L'algorithme commence donc par une recherche globale, et va trouver des zones plus intéressantes. Puis lorsque la température décroît, il va se concentrer de plus en plus sur une seule zone, et se termine comme une recherche de gradient classique. Les probabilités de trouver l'optimum global et non un optimum local sont donc plus fortes.



À chaque pas de temps, on teste donc une solution voisine à la solution actuelle. Si elle améliore les résultats, on la garde. Si au contraire, elle est moins bonne, on la garde avec une probabilité dépendant de la température. On choisit pour cette probabilité un calcul généralement basé sur une exponentielle, appelée "**règle de Metropolis**", mais des variantes sont tout à fait possibles.

Cette règle dit que la probabilité d'accepter une solution moins optimale que la solution actuelle est de :

$$P = e^{-\frac{\Delta E}{T}}$$

Dans cette équation, $\Delta\mathbf{E}$ représente la perte de qualité de la solution (notée comme une différence d'énergie, par analogie avec la physique). T est la température actuelle du système. Cette exponentielle est toujours comprise entre 0 et 1, et plus la température décroit, plus la fraction est haute et donc plus la probabilité est faible.

Les difficultés de cet algorithme se situent dans les choix des paramètres. En effet, il est important de choisir la température initiale et la loi de décroissance de celle-ci : si la température décroit de manière trop importante, l'algorithme peut ne pas avoir le temps de converger. Au contraire, si la température ne décroit pas assez vite, l'algorithme peut en permanence sortir des zones qu'il explore pour en explorer d'autres sans jamais trouver d'optimums.

La seule façon de choisir ces paramètres reste encore une fois la méthode empirique.

7. Optimisation par essaims particulaires

Pour la plupart des métaheuristiques, les résultats sont meilleurs si on lance plusieurs exécutions à partir de solutions initiales différentes. En effet, cela permet de parcourir une plus grande zone de recherche.

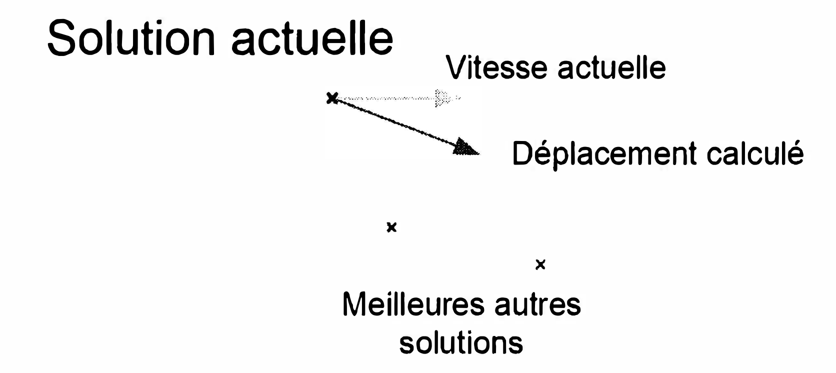
Cependant, il est possible de retomber plusieurs fois sur la même solution, et de passer à côté de l'optimum global (ou d'un meilleur optimum local).

L'**optimisation par essaims particulaires** s'inspire de la biologie. En effet, autant chez les oiseaux que chez les poissons, on peut observer de grands groupes d'animaux se déplaçant ensemble en trois dimensions. Les oiseaux (ou les poissons) ne se rentrent cependant pas dedans : la direction de chacun s'adapte en permanence en fonction de la direction actuelle et de la position des autres. Sa vitesse aussi s'adapte.

Dans cet algorithme, plusieurs solutions potentielles cohabitent dans l'espace de recherche, et chacune se déplace dans une direction donnée. À chaque itération, les solutions vont se déplacer comme une nuée, en allant vers les zones qui semblent les plus intéressantes.

Chaque solution doit connaître sa vélocité actuelle, sous la forme d'un vecteur (ce qui permet d'indiquer la direction du déplacement) et les meilleures positions découvertes jusqu'alors. De plus, toutes les solutions de l'essaim connaissent la meilleure solution actuelle (et son emplacement).

Des équations simples (à définir selon les problèmes) permettent de mettre à jour la vitesse de déplacement puis la position de la solution en fonction des différents attributs (vitesse et meilleures solutions).



On peut noter que cette métaheuristique, contrairement aux précédentes, n'utilise pas du tout la dérivée entre la solution actuelle et son voisinage. Cela permet à l'optimisation par essaims particulaires de s'appliquer à davantage de problèmes.

Cependant, encore une fois, le choix des paramètres est primordial. S'ils sont mal choisis, la convergence se fait sur le premier optimum local découvert. Au contraire, on peut observer les individus se déplacer en permanence d'une zone à une autre sans jamais converger. Il faut ainsi trouver le bon équilibre entre l'**exploration** (pour laisser l'essaim découvrir d'autres zones) et l'**exploitation** (pour chercher l'optimum local de la zone en cours).

Comme pour les autres algorithmes, ce choix se fait de manière empirique.

De plus, il existe un biais lors de la recherche : les optimums au centre de l'espace sont plus faciles à trouver. En effet, les vitesses sont mises à jour dimension par dimension. Il peut donc être opportun de transformer l'espace pour éviter ce biais ou au contraire s'en servir pour améliorer les résultats.

8. Méta-optimisation

Comme la recherche des paramètres des métaheuristiques reste un problème complexe lui aussi, on peut tout à fait imaginer utiliser un algorithme d'optimisation pour cette recherche.

Lorsque les paramètres sont découverts via une recherche d'optimum, on parle alors de **méta-optimisation** : l'optimisation du processus d'optimisation lui-même.

On peut utiliser les différentes métaheuristiques dont il est fait l'objet dans ce chapitre, mais on peut aussi utiliser d'autres techniques comme un système expert (qui contiendrait des règles issues de l'expérience de chercheurs) ou des algorithmes génétiques, voire même des réseaux de neurones.

Les différentes techniques ne sont donc pas indépendantes et peuvent être utilisées pour se compléter et s'améliorer.

9. Domaines d'application

Ces algorithmes sont très utiles dans de nombreux domaines, en particulier ceux pour lesquels il n'existe pas de moyen de calculer l'optimum de manière mathématique ou lorsque cela prendrait trop de temps.

Ils obtiennent un optimum, local ou global. On espère alors avoir un résultat, s'il n'est pas global, au moins le plus proche de celui-ci au niveau de sa qualité.

On les retrouve ainsi dans tous les domaines nécessitant la **conception** de pièces ou de systèmes. En effet, ils permettent de trouver facilement des formes ou des matériaux adéquats, en limitant le coût (ou, selon les problèmes, la surface de frottement, les turbulences...). Ils sont utilisés en **construction** par exemple, pour optimiser les structures porteuses.

Des études ont ainsi été faites pour optimiser le coût de structures en fer pour des constructions devant respecter les normes antisismiques.

En **électronique**, on s'en sert pour améliorer le design de cartes imprimées, en limitant par exemple la quantité de "fil" nécessaire, ou en minimisant la place requise par les différents composants.

En **finance**, les métaheuristiques peuvent permettre d'optimiser un portefeuille d'actions, en limitant les risques et en cherchant à maximiser les gains pour une somme donnée.

Ils sont utilisés dans les problèmes d'**ordonnancement** comme par exemple créer des horaires de bus/avions/trains. On cherche alors à minimiser le coût pour l'entreprise et à maximiser les gains. Pour cela, on cherche à laisser les véhicules le moins possible en gare (ou aéroport).

L'ordonnancement et la **planification** ne concernent pas que les horaires. Il peut aussi s'agir de production de biens en fonction des matières premières et des ressources nécessaires, ou au contraire de savoir quand et quoi commander pour mener à leurs termes les productions prévues.

Les **militaires** s'en servent pour assigner des moyens de défense à un ensemble d'attaques. L'algorithme d'optimisation permet d'aider l'opérateur humain, qui est le seul à pouvoir donner son feu vert en cas de crise. De cette façon, il est plus facile de gérer les troupes et armes pour faire face à des attaques spécifiques sur plusieurs fronts.

En **télécommunications**, ils peuvent servir à améliorer le routage ou la création de réseaux, en optimisant les temps de transfert ou la quantité de fils nécessaires pour relier les différentes machines, tout en respectant des contraintes de sécurité.

Les applications sont donc nombreuses et variées, car les métaheuristiques sont très adaptables et donnent de bons résultats, avec un coût de mise en œuvre faible.

10. Implémentation

Dans un premier temps, les algorithmes génériques sont implémentés, puis des classes héritant de ces classes-mères permettent de résoudre le problème du sac à dos.

Deux versions du problème du sac à dos sont utilisées : la première est celle présentée comme exemple pour l'algorithme glouton (avec 16 objets) et la deuxième est une version plus complexe et aléatoire, qui permet de mieux comparer les différents algorithmes.

Une analyse des résultats obtenus est menée à la fin de cette partie.

10.1 Classes génériques

Il faut commencer par définir quelques classes ou interfaces très génériques. Celles-ci nous permettent de créer ensuite les différents algorithmes.

ISolution est une interface qui représente une solution potentielle à un problème donné. La seule obligation pour cette solution est d'avoir une propriété permettant de connaître sa valeur.

```
public interface Isolution
{
    double Value { get; }
}
```

Il est alors possible de définir un problème, là encore grâce à une interface **IProblem**. On doit pouvoir alors obtenir une solution aléatoire (`RandomSolution()`), le voisinage d'une solution (`Neighbourhood`) et enfin la meilleure solution dans une liste fournie. Toutes ces méthodes sont utilisées par plusieurs algorithmes.

```
using System.Collections.Generic;

public interface Iproblem
{
    List<ISolution> Neighbourhood(ISolution _currentSolution);

    ISolution RandomSolution();
```

```
    ISolution BestSolution(List<ISolution> _neighbours);
}
```

De manière à avoir un code le plus générique possible, nous allons aussi séparer l'interface **IHM** du reste du programme. De cette façon, le code fourni est disponible pour toutes les plateformes sans modification. Le programme principal, lui, est une application en mode console pour Windows, mais on pourrait facilement l'adapter pour d'autres supports. La seule méthode nécessaire permet d'afficher un message fourni en paramètre.

```
using System;

public interface IHM
{
    void PrintMessage(String _message);
}
```

Algorithm est la dernière classe générique. Elle ne possède que deux méthodes : l'une demandant de résoudre un problème et l'autre permettant de créer le résultat de l'algorithme. De plus, on a deux attributs : l'un permettant d'avoir un lien vers le problème à résoudre et l'autre vers la classe servant d'interface avec l'utilisateur.

```
public abstract class Algorithm
{
    protected Problem pb;
    protected IHM ihm;
    public virtual void Solve(Problem _pb, IHM _ihm)
    {
           pb = _pb;
           ihm = _ihm;
    }

    protected abstract void SendResult();
}
```

10.2 Implémentation des différents algorithmes

Les classes génériques étant codées, nous pouvons passer aux différents algorithmes. Ceux-ci sont ici génériques, et il s'agit donc de classes abstraites. Il faudra créer des classes filles pour chaque algorithme et pour chaque problème : les métaheuristiques sont généralisables et il faut les adapter.

De plus, chaque classe possède une méthode `Solve` qui est un pattern de méthode : cette méthode est scellée, ce qui veut dire que les descendants ne peuvent pas la redéfinir. Par contre, elle utilise des méthodes abstraites, qui elles, devront être redéfinies. De cette façon, l'algorithme global est fixé, mais les détails de l'implémentation sont à la charge des classes filles.

10.2.1 Algorithme glouton

L'algorithme glouton **GreedyAlgorithm** est le plus simple : nous construisons une solution, morceau par morceau, et c'est cette solution que nous allons afficher. Nous avons donc uniquement besoin d'une méthode `ConstructSolution`.

```
public abstract class GreedyAlgorithm : Algorithm
{
    public override sealed void Solve(IProblem _pb, IHM _ihm)
    {
           base.Solve(_pb, _ihm);
           ConstructSolution();
           SendResult();
    }

    protected abstract void ConstructSolution();
}
```

10.2.2 Descente de gradient

L'algorithme suivant est **GradientDescentAlgorithm**, pour la descente de gradient. L'algorithme général consiste à créer une première solution aléatoire, puis, tant qu'un critère d'arrêt n'est pas atteint, on demande le voisinage d'une solution, et si ce voisinage existe, on choisit la meilleure solution à l'intérieur. On lance alors la mise à jour de la solution en cours via `UpdateSolution` (qui change ou non la solution, selon la présence ou l'absence d'une amélioration).

La boucle se termine par l'incrémentation des différentes variables permettant d'atteindre le critère d'arrêt. Ensuite les résultats sont affichés.

On a donc besoin des méthodes suivantes :

- `Done()` : indique si oui ou non les critères d'arrêt sont atteints.
- `UpdateSolution()` : met à jour (ou non) la solution actuelle, selon la nouvelle solution proposée en paramètre.
- `Increment()` : met à jour les critères d'arrêt à chaque boucle.

```
using System.Collections.Generic;

public abstract class GradientDescentAlgorithm : Algorithm
{
    protected ISolution currentSolution;

    public override sealed void Solve(IProblem _pb, IHM _ihm)
    {
           base.Solve(_pb, _ihm);

           currentSolution = pb.RandomSolution();
           while (!Done())
           {
                  List<ISolution> Neighbours =
pb.Neighbourhood(currentSolution);
                  if (Neighbours != null)
                  {
                              ISolution bestSolution =
pb.BestSolution(Neighbours);
                              UpdateSolution(bestSolution);
                  }
                  Increment();
           }
           SendResult();
   }

   protected abstract bool Done();

   protected abstract void UpdateSolution(ISolution _bestSolution);

   protected abstract void Increment();
}
```

10.2.3 Recherche tabou

La recherche tabou **TabuSearchAlgorithm** est plus complexe : on va garder la meilleure solution trouvée jusqu'ici et la solution en cours.

On commence par initialiser la solution de départ aléatoirement. Ensuite, tant que les critères d'arrêt ne sont pas atteints (méthode `Done()`), le voisinage de la solution actuelle est créé. Dans celui-ci, les solutions présentes dans la liste des positions taboues sont enlevées (`RemoveSolutionsInTabuList`). On garde alors la meilleure solution avec sa mise à jour (ou non) grâce à la méthode `UpdateSolution`.

Il ne reste alors qu'à incrémenter les compteurs (`Increment`), puis à la fin de la boucle, afficher le résultat.

```
using System.Collections.Generic;

public abstract class TabuSearchAlgorithm : Algorithm
{
    protected ISolution currentSolution;

    protected ISolution bestSoFarSolution;

    public override sealed void Solve(IProblem _pb, IHM _ihm)
    {
            base.Solve(_pb, _ihm);

            currentSolution = pb.RandomSolution();
            bestSoFarSolution = currentSolution;
            AddToTabuList(currentSolution);

            while (!Done())
            {
                    List<ISolution> Neighbours =
pb.Neighbourhood(currentSolution);
                    if (Neighbours != null)
                    {
                            Neighbours =
RemoveSolutionsInTabuList(Neighbours);
                            ISolution bestSolution =
pb.BestSolution(Neighbours);
                            if (bestSolution != null)
                            {
                                    UpdateSolution(bestSolution);
                            }
```

```
                }
                Increment();
            }
            SendResult();
        }

        protected abstract void AddToTabuList(ISolution
currentSolution);

        protected abstract List<ISolution>
RemoveSolutionsInTabuList(List<ISolution> Neighbours);

        protected abstract bool Done();

        protected abstract void UpdateSolution(ISolution _bestSolution);

        protected abstract void Increment();
}
```

10.2.4 Recuit simulé

Dans le cas du recuit simulé **SimulatedAnnealingAlgorithm**, on repart de la descente de gradient. On garde cependant la meilleure solution trouvée jusqu'alors, car on peut accepter une solution moins performante.

Il faut donc commencer par l'initialisation de la température (InitTemperature). On boucle ensuite jusqu'à ce que les critères d'arrêt soient atteints (Done). À chaque itération, le voisinage de la solution en cours est récupéré, et en particulier la meilleure solution parmi celui-ci. Puis on met à jour (ou non) cette dernière (UpdateSolution). C'est dans cette méthode que la température est utilisée.

La boucle se finit par l'incrément des variables internes (Increment) et la modification de la température (UpdateTemperature). L'algorithme se termine par l'affichage des résultats.

```
using System.Collections.Generic;

public abstract class SimulatedAnnealingAlgorithm : Algorithm
{
    protected ISolution currentSolution;
    protected ISolution bestSoFarSolution;
    protected double temperature;
```

```
    public override sealed void Solve(IProblem _pb, IHM _ihm)
    {
          base.Solve(_pb, _ihm);

          currentSolution = pb.RandomSolution();
          bestSoFarSolution = currentSolution;

          InitTemperature();
          while (!Done())
          {
                List<ISolution> Neighbours =
pb.Neighbourhood(currentSolution);
                if (Neighbours != null)
                {
                      ISolution bestSolution =
pb.BestSolution(Neighbours);
                      UpdateSolution(bestSolution);
                }
                Increment();
                UpdateTemperature();
          }
          SendResult();
    }

    protected abstract void UpdateTemperature();

    protected abstract void InitTemperature();

    protected abstract bool Done();

    protected abstract void UpdateSolution(ISolution _bestSolution);

    protected abstract void Increment();
}
```

10.2.5 Optimisation par essaims particulaires

Le dernier algorithme est l'optimisation par essaims particulaires **`ParticleSwarmOptimizationAlgorithm`**. Celui-ci est très différent des autres : en effet, au lieu d'utiliser un voisinage composé de plusieurs solutions et de n'en retenir qu'une, on a une population de solutions (`currentSolutions`) qui vont se déplacer dans l'environnement. En plus de la meilleure solution rencontrée jusqu'alors, on va aussi garder la meilleure solution actuelle au sein de la population.

La constante NB_INDIVIDUALS indique le nombre d'individus utilisés dans notre population (ici 30). Il s'agit d'un des paramètres à adapter.

On commence donc par initialiser notre population aléatoirement et à mettre à jour nos meilleures solutions initiales. On boucle ensuite jusqu'à atteindre un critère d'arrêt (Done).

À chaque itération, on met à jour les meilleures solutions globales (UpdateGeneralVariables) puis les positions des solutions (UpdateSolutions) et enfin les variables utilisées pour le critère d'arrêt sont incrémentées (Increment). Une fois la boucle finie, on affiche les résultats.

```
using System;
using System.Collections.Generic;

public abstract class ParticleSwarmOptimizationAlgorithm :
Algorithm
{
    protected List<ISolution> currentSolutions;
    protected ISolution bestSoFarSolution;
    protected ISolution bestActualSolution;

    protected const int NB_INDIVIDUALS = 30;

    public override sealed void Solve(IProblem _pb, IHM _ihm)
    {
          base.Solve(_pb, _ihm);

          currentSolutions = new List<ISolution>();
          for (int i = 0; i < NB_INDIVIDUALS; i++)
          {
                 ISolution newSolution = pb.RandomSolution();
                 currentSolutions.Add(newSolution);
          }

          bestActualSolution = pb.BestSolution(currentSolutions);
          bestSoFarSolution = bestActualSolution;

          while (!Done())
          {
                 UpdateGeneralVariables();
                 UpdateSolutions();
                 Increment();
      }
```

```
            SendResult();
        }

        protected abstract void UpdateSolutions();

        protected abstract void UpdateGeneralVariables();

        protected abstract bool Done();

        protected abstract void Increment();
    }
```

10.3 Résolution du problème du sac à dos

Nous allons maintenant appliquer les métaheuristiques au problème du sac à dos (`KnapsackProblem` en anglais). Pour cela, nous allons commencer par dériver les classes de base (`ISolution` et `IProblem`), puis nous coderons les différentes méthodes nécessaires aux différents algorithmes.

10.3.1 Implémentation du problème

Avant de pouvoir coder le problème, il faut définir le contenu du sac à dos : les boîtes (**Box**). Chaque boîte a un poids et une valeur, ainsi qu'un nom qui servira surtout pour l'affichage.

Pour faciliter la création des boîtes et leur maniement, un constructeur est ajouté, permettant d'initialiser les trois propriétés. La méthode `ToString` est aussi substituée pour pouvoir afficher les informations de la boîte.

```
using System;

public class Box
{
    public double Weight { get; set; }
    public double Value { get; set; }

    String Name { get; set; }

    public Box(String _name, double _weight, double _value)
    {
          Name = _name;
          Weight = _weight;
          Value = _value;
    }

    public override string ToString()
    {
          return Name + " (" + Weight + ", " + Value + ")";
    }
}
```

Une fois les boîtes définies, il est possible de créer les solutions **KnapsackSolution**, qui implémentent l'interface ISolution.

```
using System;
using System.Collections.Generic;
using System.Linq;

public class KnapsackSolution : Isolution
{
    // Code ici
}
```

Tout d'abord, nous définissons trois propriétés :

- LoadedContent : correspond au contenu chargé dans le sac à dos et est donc une liste de boîtes.
- Weight : poids contenu dans le sac à dos. Celui-ci est recalculé à chaque fois, comme étant la somme des poids des boîtes contenues.
- Value : valeur totale du contenu du sac. Celle-ci est aussi recalculée à chaque demande.

```
public double Weight
{
      get
      {
            return LoadedContent.Sum(x => x.Weight);
      }
}

public double Value
{
      get
      {
            return LoadedContent.Sum(x => x.Value);
      }
}

List<Box> loadedContent;
public List<Box> LoadedContent
{
      get
      {
            return loadedContent;
      }
      set
      {
            loadedContent = value;
      }
}
```

Deux constructeurs sont ajoutés. Le premier est un constructeur par défaut, qui ne fait que créer une nouvelle liste vide de contenu. Le deuxième est un constructeur qui copie le contenu de la solution passée en paramètre.

```
public KnapsackSolution()
{
      loadedContent = new List<Box>();
}

public KnapsackSolution(KnapsackSolution _solution)
{
      loadedContent = new List<Box>();
      loadedContent.AddRange(_solution.loadedContent);
}
```

On redéfinit enfin les trois méthodes de base des objets : `ToString`, `Equals` (pour comparer deux sacs à dos) et `GetHashCode`. La méthode `ToString` va simplement créer une chaîne contenant la valeur, le poids puis le contenu du sac à dos.

```
    public override string ToString()
    {
            String res = "Value : " + Value + " - Weight : " +
Weight + "\n";
            res += "Loaded : ";
            res += String.Join(" - ", loadedContent);
            return res;
    }
```

Pour le comparateur d'égalité, il faut vérifier tout d'abord que les deux sacs à dos ont le même nombre d'objets, le même poids et la même valeur. Si c'est le cas, on teste alors si chaque boîte contenue dans le premier sac à dos se retrouve dans le deuxième. Il n'est pas possible de comparer directement les listes, car les objets peuvent se trouver dans un ordre différent.

```
    public override bool Equals(object _object)
    {
            KnapsackSolution solution = (KnapsackSolution)_object;
            if (solution.loadedContent.Count != loadedContent.Count ||
solution.Value != Value || solution.Weight != Weight)
            {
                    return false;
            }
            else
            {
                    foreach (Box box in loadedContent) {
                            if (!solution.loadedContent.Contains(box))
                            {
                                    return false;
                            }
                    }

            }
            return true;
    }
```

La méthode `GetHashCode` est demandée pour différencier rapidement des solutions si elles sont utilisées dans un dictionnaire par exemple. Malheureusement, aucune des propriétés actuelles de la solution n'est fixe. On renvoie ici la multiplication de la valeur par le poids, mais si les solutions devaient être modifiées après l'ajout dans un dictionnaire, cela ne fonctionnerait plus correctement.

```
    public override int GetHashCode()
    {
          return (int) (Value * Weight);
    }
```

On termine par l'implémentation du problème du sac à dos lui-même **`KnapsackProblem`**, qui implémente `IProblem`.

Cet objet contient une liste de boîtes disponibles à mettre dans les sacs à dos nommées `boxes`, ainsi qu'un poids maximal pouvant être chargé (sous la forme d'une propriété), un générateur aléatoire rendu statique et enfin une constante indiquant le nombre de voisins que l'on créera pour chaque solution. Ce nombre peut être modifié en fonction des besoins.

```
using System;
using System.Collections.Generic;
using System.Linq;

public class KnapsackProblem : Iproblem
{
    protected List<Box> boxes = null;
    public double MaxWeight { get; set; }

    public static Random randomGenerator = null;

    public const int NB_NEIGHBOURS = 30;

    // Reste du code ici
}
```

Deux constructeurs sont disponibles :

– Le constructeur par défaut construit l'exemple présenté au début de ce chapitre, avec un sac à dos d'une contenance de 20 kg, et 12 boîtes utilisables.

- Un deuxième constructeur construit un nouveau problème. On lui donne alors le nombre de boîtes disponibles, la taille du sac et la valeur maximale de chaque boîte. Celles-ci sont ensuite créées aléatoirement.

```
    public KnapsackProblem()
    {
            boxes = new List<Box>();

            boxes.Add(new Box("A", 4, 15));
            boxes.Add(new Box("B", 7, 15));
            boxes.Add(new Box("C", 10, 20));
            boxes.Add(new Box("D", 3, 10));
            boxes.Add(new Box("E", 6, 11));
            boxes.Add(new Box("F", 12, 16));
            boxes.Add(new Box("G", 11, 12));
            boxes.Add(new Box("H", 16, 22));
            boxes.Add(new Box("I", 5, 12));
            boxes.Add(new Box("J", 14, 21));
            boxes.Add(new Box("K", 4, 10));
            boxes.Add(new Box("L", 3, 7));

            MaxWeight = 20;
            if (randomGenerator == null)
            {
                    randomGenerator = new Random();
            }
    }

    public KnapsackProblem(int _nbBoxes, double _maxWeight,
double _maxValue)
    {
            boxes = new List<Box>();
            MaxWeight = _maxWeight;
            if (randomGenerator == null)
            {
                    randomGenerator = new Random();
            }

            for (int i = 0; i < _nbBoxes; i++)
            {
                    boxes.Add(new Box(i.ToString(),
randomGenerator.NextDouble() * _maxWeight,
randomGenerator.NextDouble() * _maxValue));
            }
    }
```

Une méthode `Boxes` est ajoutée pour renvoyer la liste des boîtes disponibles.

```
public List<Box> Boxes() {
        return boxes;
}
```

Il faut enfin implémenter les trois méthodes de l'interface `IProblem`. La première consiste à créer aléatoirement une nouvelle solution. Pour cela, on tire au sort des boîtes une à une parmi les boîtes disponibles et non encore utilisées (grâce à l'utilisation de Linq et des méthodes `Except` et `Where`). Pour s'assurer qu'il s'agit d'une solution viable, on vérifie que l'espace disponible est suffisant. Lorsqu'on ne peut plus ajouter de boîtes au sac à dos, on renvoie la solution créée.

```
public ISolution RandomSolution()
{
      KnapsackSolution solution = new KnapsackSolution();
      List<Box> possibleBoxes = boxes;

      double enableSpace = MaxWeight;
      List<Box> availableBoxes = possibleBoxes.Where(x =>
(x.Weight <= enableSpace)).ToList();

      while (enableSpace > 0 && availableBoxes.Count != 0)
      {
            int index = randomGenerator.Next(0, availableBoxes.Count);

solution.LoadedContent.Add(availableBoxes.ElementAt(index));
            enableSpace = MaxWeight - solution.Weight;
            availableBoxes =
possibleBoxes.Except(solution.LoadedContent).Where(x => (x.Weight
<= enableSpace)).ToList();
      }

      return solution;
}
```

La deuxième consiste à choisir la meilleure solution dans une liste. Pour nous, il s'agit simplement de chercher la solution avec la valeur maximale. On pourrait utiliser la commande Linq `OrderBy`, mais celle-ci est très peu efficace car elle demande de trier toute la liste. À la place, on va chercher quelle est la valeur maximale des boîtes (ce qui est assez rapide) puis on prend la première solution avec cette valeur.

```
    public ISolution BestSolution(List<ISolution> _listSolutions)
    {
            return _listSolutions.Where(x =>
x.Value.Equals(_listSolutions.Max(y =>
y.Value))).FirstOrDefault();
    }
```

La dernière méthode, `Neighbourhood`, consiste à renvoyer le voisinage de la solution passée en paramètre. Pour cela, on réalise de légères modifications à la solution : on lui enlève un objet pris au hasard, puis on complète l'espace ainsi libéré par d'autres boîtes aléatoirement. On recommence autant de fois que de voisins souhaités.

```
    public List<ISolution> Neighbourhood(ISolution _currentSolution)
    {
          List<ISolution> neighbours = new List<ISolution>();
          List<Box> possibleBoxes = boxes;

          for (int i = 0; i < NB_NEIGHBOURS; i++)
          {
                 KnapsackSolution newSol = new
KnapsackSolution((KnapsackSolution)_currentSolution);
                 int index = randomGenerator.Next(0,
newSol.LoadedContent.Count);
                 newSol.LoadedContent.RemoveAt(index);

                 double enableSpace = MaxWeight - newSol.Weight;
                 List<Box> availableBoxes =
possibleBoxes.Except(newSol.LoadedContent).Where(x => (x.Weight
<= enableSpace)).ToList();

                 while (enableSpace > 0 && availableBoxes.Count != 0)
                 {
                        index = randomGenerator.Next(0,
availableBoxes.Count);

   newSol.LoadedContent.Add(availableBoxes.ElementAt(index));
```

```
                        enableSpace = MaxWeight - newSol.Weight;
                        availableBoxes =
possibleBoxes.Except(newSol.LoadedContent).Where(x => (x.Weight
<= enableSpace)).ToList();
                    }

                    neighbours.Add(newSol);
                }
                return neighbours;
    }
```

Le problème du sac à dos est maintenant complètement codé. Il ne reste plus qu'à dériver les différents algorithmes.

10.3.2 Algorithme glouton

Pour implémenter les différents algorithmes, il n'y a plus besoin de coder l'algorithme (qui est dans la méthode `Solve` des classes déjà codées) mais uniquement les méthodes implémentant les détails.

Pour l'algorithme glouton, la classe **`GreedyAlgorithmForKnapsack`** a seulement deux méthodes à créer : une permettant de construire la solution et une permettant d'afficher le résultat.

Cette classe possède donc un seul attribut qui est la solution en cours de construction. La méthode `SendResult` consiste simplement à l'afficher.

Pour la méthode `ConstructSolution`, on demande tout d'abord la liste des boîtes utilisables. On les trie ensuite du plus haut au plus faible rapport "valeur/poids". Tant qu'il existe de l'espace dans le sac à dos, on rajoute ces boîtes.

```
using System.Collections.Generic;
using System.Linq;

public class GreedyAlgorithmForKnapsack : GreedyAlgorithm
{
    KnapsackSolution solution;

    protected override void ConstructSolution()
    {
        KnapsackProblem problem = (KnapsackProblem) pb;
        List<Box> boxes = problem.Boxes();
```

```
            solution = new KnapsackSolution();

            foreach(Box currentBox in boxes.OrderByDescending(x =>
x.Value / x.Weight))
            {
                  double spaceLeft = problem.MaxWeight -
solution.Weight;
                  if (currentBox.Weight < spaceLeft)
                  {
                        solution.LoadedContent.Add(currentBox);
                  }
            }
      }

      protected override void SendResult()
      {
            ihm.PrintMessage(solution.ToString());
      }
}
```

10.3.3 Descente de gradient

Pour la descente de gradient, la classe **GradientDescentForKnapsack** doit contenir quatre méthodes.

Le critère d'arrêt est le nombre d'itérations sans avoir trouvé d'amélioration. En effet, le voisinage étant créé aléatoirement, il va falloir en tester plusieurs pour savoir s'il y a ou non des améliorations possibles autour du point actuel. On fixe le nombre maximal d'itérations à 50.

```
public class GradientDescentForKnapsack : GradientDescentAlgorithm
{
    int nbIterationsWithoutUpdate = 0;
    private const int MAX_ITERATIONS_WITHOUT_UPDATE = 50;


    // Méthodes ici
}
```

La première méthode est Done, qui doit indiquer quand le critère d'arrêt a été atteint. Il suffit donc de vérifier si on a atteint la limite du nombre d'itérations sans amélioration.

```
    protected override bool Done()
    {
          return nbIterationsWithoutUpdate >=
MAX_ITERATIONS_WITHOUT_UPDATE;
    }
```

La deuxième méthode est Increment, qui doit incrémenter les différentes variables internes. Ici nous n'avons que le nombre d'itérations sans amélioration à incrémenter.

```
    protected override void Increment()
    {
           nbIterationsWithoutUpdate++;
    }
```

La troisième méthode est celle qui doit mettre à jour (ou non) une solution en la remplaçant par celle passée en paramètre. Pour cela, on regarde simplement si la valeur est plus forte que celle en cours. Si oui, on remplace et on remet à 0 le compteur indiquant le nombre d'itérations sans amélioration.

```
    protected override void UpdateSolution(ISolution
 _bestSolution)
    {
            if (_bestSolution.Value > currentSolution.Value)
            {
                   currentSolution = _bestSolution;
                   nbIterationsWithoutUpdate = 0;
            }
    }
```

La dernière méthode consiste simplement à envoyer le résultat, à savoir la solution en cours.

```
    protected override void SendResult()
    {
           ihm.PrintMessage(currentSolution.ToString());
    }
```

10.3.4 Recherche tabou

La recherche tabou est plus complexe. En effet, elle nécessite d'avoir une liste des positions taboues, cette liste étant de taille fixe. De plus, il faut fixer un critère d'arrêt plus complexe, pour éviter de boucler entre plusieurs positions.

On choisit donc un double critère d'arrêt : tout d'abord, on compte le nombre d'itérations pendant lesquelles on ne trouve plus d'améliorations, en se limitant à `MAX_ITERATIONS_WITHOUT_UPDATE`, constante qui est fixée préalablement. De plus, on s'arrête aussi au bout d'un nombre d'itérations fixées `MAX_ ITERATIONS`.

Notre classe **`TabuSearchForKnapsack`** possède donc trois attributs (les deux nombres d'itérations et la liste des positions taboues) et trois constantes.

```
using System.Collections.Generic;
using System.Linq;

public class TabuSearchForKnapsack : TabuSearchAlgorithm
{
    int nbIterationsWithoutUpdate = 1;
    int nbIterations = 1;
    private const int MAX_ITERATIONS_WITHOUT_UPDATE = 30;
    private const int MAX_ITERATIONS = 100;
    private const int TABU_SEARCH_MAX_SIZE = 50;

    List<KnapsackSolution> tabuSolutions = new
List<KnapsackSolution>();

    // Autres méthodes ici
}
```

Cette classe implémentant la classe `TabuSearchAlgorithm`, il faut maintenant développer les six méthodes abstraites. La première consiste à indiquer si le critère d'arrêt est atteint. Pour cela, on vérifie si on a dépassé nos deux nombres maximums d'itérations.

```
   protected override bool Done()
   {
          return nbIterationsWithoutUpdate >=
MAX_ITERATIONS_WITHOUT_UPDATE && nbIterations >= MAX_ITERATIONS;
   }
```

La deuxième méthode consiste à mettre à jour (ou non) la solution actuelle par la solution passée en paramètre. Pour cela, on vérifie simplement si la meilleure solution proposée est contenue dans la liste des positions taboues. Si non, on met à jour la position et on l'ajoute aux positions taboues. De plus, si on a atteint une solution meilleure que celle obtenue jusqu'alors, on met à jour la variable `bestSoFarSolution`. Enfin, on remet à zéro le nombre d'itérations sans mises à jour.

```
    protected override void UpdateSolution(ISolution _bestSolution)
    {
            if
(!tabuSolutions.Contains((KnapsackSolution)_bestSolution))
            {
                currentSolution = _bestSolution;
                AddToTabuList((KnapsackSolution)_bestSolution);
                if (_bestSolution.Value >
bestSoFarSolution.Value)
            {
                    bestSoFarSolution = _bestSolution;
                    nbIterationsWithoutUpdate = 0;
                }
            }
    }
```

La méthode suivante est celle permettant de mettre à jour les variables internes. Nous incrémentons donc les deux variables correspondant aux nombres d'itérations.

```
    protected override void Increment()
    {
          nbIterationsWithoutUpdate++;
          nbIterations++;
    }
```

La quatrième est celle permettant l'affichage de la meilleure solution.

```
    protected override void SendResult()
    {
          ihm.PrintMessage(bestSoFarSolution.ToString());
    }
```

La cinquième méthode ajoute la position donnée à la liste des positions taboues. Pour cela, on vérifie d'abord si on a atteint le nombre maximal de positions, et si oui, on enlève la toute première de la liste avant d'ajouter la nouvelle position.

```
protected override void AddToTabuList(ISolution _solution)
{
        while (tabuSolutions.Count >= TABU_SEARCH_MAX_SIZE)
        {
                tabuSolutions.RemoveAt(0);
        }
        tabuSolutions.Add((KnapsackSolution)_solution);
}
```

Enfin, la dernière méthode consiste simplement à renvoyer les positions d'un voisinage qui ne sont pas dans la liste taboue. Pour cela, on utilise la commande Linq `Except`.

```
protected override List<ISolution>
RemoveSolutionsInTabuList(List<ISolution> Neighbours)
{
        return Neighbours.Except(tabuSolutions).ToList();
}
```

La recherche tabou est maintenant opérationnelle.

10.3.5 Recuit simulé

Le recuit simulé doit accepter des solutions moins bonnes que la solution actuelle, avec une probabilité dépendant de la température en cours.

La classe **`SimulatedAnnealingForKnapsack`** hérite donc de la classe abstraite `SimulatedAnnealingAlgorithm`. Seuls les détails de l'implémentation adaptés au problème doivent donc être codés.

Tout d'abord, nous aurons besoin comme pour la recherche tabou, de deux indicateurs pour l'arrêt de l'algorithme : le nombre d'itérations depuis le début, et le nombre d'itérations sans avoir rencontré d'amélioration. Ces deux nombres sont limités par des constantes définies dans le code.

```
using System;
using System.Collections.Generic;
using System.Linq;

public class SimulatedAnnealingForKnapsack :
SimulatedAnnealingAlgorithm
{
    int nbIterationsWithoutUpdate = 1;
    int nbIterations = 1;
    private const int MAX_ITERATIONS_WITHOUT_UPDATE = 30;
    private const int MAX_ITERATIONS = 100;

    // Autres méthodes ici
}
```

Il faut ensuite implémenter les différentes méthodes abstraites de la classe mère. La première consiste à mettre à jour la température à chaque itération. On applique simplement une multiplication par 0.9, ce qui permet de baisser celle-ci graduellement.

```
    protected override void UpdateTemperature()
    {
           temperature *= 0.9;
    }
```

Pour l'initialisation de la température, on part de 5. Cette valeur doit être adaptée en fonction des problèmes, et la seule méthode est empirique.

```
    protected override void InitTemperature()
    {
           temperature = 5;
    }
```

La troisième méthode est celle permettant d'arrêter l'algorithme. On vérifie donc que les deux compteurs sont inférieurs aux valeurs maximales.

```
    protected override bool Done()
    {
           return nbIterationsWithoutUpdate >=
MAX_ITERATIONS_WITHOUT_UPDATE && nbIterations >= MAX_ITERATIONS;
    }
```

La méthode suivante est la plus complexe de cette classe : elle permet de savoir s'il faut ou non mettre à jour la solution actuelle. Pour cela, on commence par regarder s'il s'agit d'une solution entraînant une perte de qualité. Si oui, on calcule la probabilité de l'accepter grâce à la loi de Metropolis. Ensuite, si on tire un nombre aléatoire inférieur à cette probabilité (ou si la solution proposée est meilleure), alors on met à jour la solution actuelle. De plus, si c'est la meilleure rencontrée jusqu'à présent, on met à jour la variable `bestSoFarSolution` et on réinitialise le nombre d'itérations sans amélioration.

```
    protected override void UpdateSolution(ISolution _bestSolution)
    {
            double seuil = 0.0;
            if (_bestSolution.Value < currentSolution.Value)
            {
                    seuil = Math.Exp(-1 * (currentSolution.Value -
_bestSolution.Value) / currentSolution.Value / temperature);
            }
            if (_bestSolution.Value > currentSolution.Value ||
KnapsackProblem.randomGenerator.NextDouble() < seuil)
            {
                    currentSolution = _bestSolution;
                    if (_bestSolution.Value > bestSoFarSolution.Value)
                    {
                            bestSoFarSolution = _bestSolution;
                            nbIterationsWithoutUpdate = 0;
                    }
            }
    }
```

La cinquième méthode incrémente simplement les deux compteurs.

```
    protected override void Increment()
    {
            nbIterationsWithoutUpdate++;
            nbIterations++;
    }
```

La sixième et dernière méthode lance l'affichage de la meilleure solution rencontrée jusqu'à présent.

```
    protected override void SendResult()
    {
            ihm.PrintMessage(bestSoFarSolution.ToString());
    }
```

Le recuit simulé est lui aussi maintenant implémenté.

10.3.6 Optimisation par essaims particulaires

Le dernier algorithme à adapter est l'optimisation par essaims particulaires. Au lieu de choisir la meilleure solution d'un voisinage, nous devons faire évoluer toutes nos solutions au cours du temps, en fonction de la meilleure solution rencontrée jusqu'à présent et de la meilleure solution en cours.

Le critère d'arrêt est seulement le nombre d'itérations. Nous avons donc un attribut comptant les itérations depuis le départ et une constante fixant le nombre d'itérations à effectuer. La base de notre classe **ParticleSwarmOptimizationForKnapsack** est donc la suivante :

```
using System.Collections.Generic;
using System.Linq;

public class ParticleSwarmOptimizationForKnapsack :
ParticleSwarmOptimizationAlgorithm
{
    int nbIterations = 1;
    private const int MAX_ITERATIONS = 200

    // Autres méthodes ici
}
```

La classe abstraite `ParticleSwarmOptimizationAlgorithm` contient cinq méthodes abstraites qu'il faut implémenter. La première, et la plus complexe, consiste à mettre à jour les différentes solutions. Pour cela, la liste de toute la population de solutions est parcourue.

Pour chacune, s'il ne s'agit pas de la meilleure trouvée jusqu'à présent, on va lui ajouter un élément de cette dernière, et un élément de la meilleure de la population actuelle. Les éléments tirés au sort ne sont ajoutés que s'ils ne sont pas déjà présents dans le sac à dos.

Après cet ajout, le sac peut avoir un poids trop important. On élimine alors aléatoirement des boîtes jusqu'à repasser sous la limite du poids. Enfin, si cela est possible, on complète le sac.

En effet, si on est à 21 kg et qu'on enlève un élément de 7 kg, on passe à 14 kg. On est bien sous la limite des 20 kg, mais on peut augmenter la valeur du sac en ajoutant jusqu'à 6 kg d'éléments.

```
    protected override void UpdateSolutions()
    {
            List<Box> possibleBoxes = ((KnapsackProblem)pb).Boxes();

            foreach (ISolution genericSolution in currentSolutions)
            {
                    KnapsackSolution solution =
(KnapsackSolution)genericSolution;

                    if (!solution.Equals(bestSoFarSolution))
                    {
                            // Ajout d'un élément de la meilleure
solution actuelle
                            int index =
KnapsackProblem.randomGenerator.Next(0,
((KnapsackSolution)bestActualSolution).LoadedContent.Count);
                            Box element =
((KnapsackSolution)bestActualSolution).LoadedContent.ElementAt(index);
                            if
(!solution.LoadedContent.Contains(element))
                            {
    solution.LoadedContent.Add(element);


                            // Ajout d'un élément de la meilleure
solution jusqu'à préent
                            index =
KnapsackProblem.randomGenerator.Next(0,
((KnapsackSolution)bestSoFarSolution).LoadedContent.Count);
                            element =
((KnapsackSolution)bestSoFarSolution).LoadedContent.ElementAt(index);
                            if
(!solution.LoadedContent.Contains(element))
                            {

    solution.LoadedContent.Add(element);
                            }

                            // On passe sous la limite de poids
                            while (solution.Weight >
((KnapsackProblem)pb).MaxWeight)
                            {
                                    index =
```

```
KnapsackProblem.randomGenerator.Next(0,
solution.LoadedContent.Count);

   solution.LoadedContent.RemoveAt(index);
                        }

                        // On complète le sac
                        double enableSpace =
((KnapsackProblem)pb).MaxWeight - solution.Weight;
                        List<Box> availableBoxes =
possibleBoxes.Except(solution.LoadedContent).Where(x => (x.Weight
<= enableSpace)).ToList();

                        while (enableSpace > 0 &&
availableBoxes.Count != 0)
                        {
                               index =
KnapsackProblem.randomGenerator.Next(0, availableBoxes.Count);

   solution.LoadedContent.Add(availableBoxes.ElementAt(index));
                               enableSpace =
((KnapsackProblem)pb).MaxWeight - solution.Weight;
                               availableBoxes =
possibleBoxes.Except(solution.LoadedContent).Where(x => (x.Weight
<= enableSpace)).ToList();
                        }
                }
          }
   }
```

La deuxième méthode doit mettre à jour les meilleures solutions. On doit donc chercher celle dans la population actuelle, et si elle est meilleure que la meilleure jusqu'à présent, on modifie aussi bestSoFarSolution.

```
   protected override void UpdateGeneralVariables()
   {
          bestActualSolution =
currentSolutions.OrderByDescending(x => x.Value).FirstOrDefault();
          if (bestActualSolution.Value > bestSoFarSolution.Value)
          {
                 bestSoFarSolution = bestActualSolution;
          }
   }
```

La troisième méthode est celle permettant de vérifier que le critère d'arrêt est ou non atteint. Il suffit donc de vérifier le nombre d'itérations.

```
protected override bool Done()
{
        return nbIterations >= MAX_ITERATIONS;
}
```

La quatrième méthode incrémente simplement le nombre d'itérations.

```
protected override void Increment()
{
        nbIterations++;
}
```

La dernière méthode est celle permettant d'afficher la meilleure solution rencontrée jusqu'à présent.

```
protected override void SendResult()
{
        ihm.PrintMessage(bestSoFarSolution.ToString());
}
```

Tous les algorithmes sont maintenant adaptés au problème du sac à dos.

10.3.7 Programme principal

Cette classe est spécifique à une application Console pour Windows. Il s'agit du programme principal, qui contient le `main`. La classe **`Program`** implémente l'interface `IHM` pour permettre les affichages. Il s'agit simplement de les écrire dans la console.

```
using MetaheuristicsPCL;
using System;

class Program : IHM
{
    public void PrintMessage(String _message)
    {
          Console.Out.WriteLine(_message);
    }

    // Autres méthodes ici (dont le main)
}
```

On crée ensuite une méthode permettant de lancer sur un problème donné en paramètre les 5 algorithmes à la suite, avec quelques affichages pour s'y retrouver.

```
private void RunAlgorithms(IProblem _pb)
{
        Algorithm algo;

        Console.Out.WriteLine("Algorithme glouton");
        algo = new GreedyAlgorithmForKnapsack();
        algo.Solve(_pb, this);
        Console.Out.WriteLine("");

        Console.Out.WriteLine("Descente de gradient");
        algo = new GradientDescentForKnapsack();
        algo.Solve(_pb, this);
        Console.Out.WriteLine("");

        Console.Out.WriteLine("Recherche tabou");
        algo = new TabuSearchForKnapsack();
        algo.Solve(_pb, this);
        Console.Out.WriteLine("");

        Console.Out.WriteLine("Recuit simulé");
        algo = new SimulatedAnnealingForKnapsack();
        algo.Solve(_pb, this);
        Console.Out.WriteLine("");

        Console.Out.WriteLine("Optimisation par essaim particulaire");
        algo = new ParticleSwarmOptimizationForKnapsack();
        algo.Solve(_pb, this);
        Console.Out.WriteLine("");
}
```

On peut alors coder une méthode `Run`, qui permet de lancer les différents algorithmes sur le problème du sac à dos simple (celui donné en exemple dans ce chapitre), puis sur un problème aléatoire plus complexe, contenant 100 boîtes d'une valeur maximale de 20, pour un sac à dos pouvant contenir jusqu'à 30 kg.

```
    protected void Run()
    {
            Console.Out.WriteLine("Métaheuristiques d'optimisation\n");

            IProblem kspb = new KnapsackProblem();
            RunAlgorithms(kspb);


    Console.Out.WriteLine("*************************************
************************\n");

            IProblem kspbRandom = new KnapsackProblem(100, 30, 20);
            RunAlgorithms(kspbRandom);

            Console.Out.WriteLine("FIN");
            while (true) ;
    }
```

Le programme principal consiste donc simplement à créer une instance de la classe `Program` et à appeler sa méthode `Run`.

```
    static void Main(string[] _args)
    {
            Program p = new Program();
            p.Run();
    }
```

Le programme est entièrement opérationnel.

10.4 Résultats obtenus

Voici un exemple de sortie obtenue tout d'abord sur le premier problème, avec la valeur et le poids du sac à dos puis son contenu, et ensuite sur le deuxième problème.

On peut voir que toutes les solutions sont identiques et correspondent à l'optimum sur le premier sac à dos.

Sur le problème aléatoire (qui change donc à chaque exécution), trois algorithmes donnent le meilleur résultat : la descente de gradient, la recherche tabou et l'optimisation par essaims particulaires.

```
Métaheuristiques d'optimisation

Algorithme glouton
Value : 54 - Weight : 19
Loaded : A (4, 15) - D (3, 10) - K (4, 10) - I (5, 12) - L (3, 7)

Descente de gradient
Value : 54 - Weight : 19
Loaded : L (3, 7) - A (4, 15) - D (3, 10) - K (4, 10) - I (5, 12)

Recherche tabou
Value : 54 - Weight : 19
Loaded : K (4, 10) - A (4, 15) - D (3, 10) - L (3, 7) - I (5, 12)

Recuit simulé
Value : 54 - Weight : 19
Loaded : I (5, 12) - K (4, 10) - A (4, 15) - D (3, 10) - L (3, 7)

Optimisation par essaim particulaire
Value : 54 - Weight : 19
Loaded : L (3, 7) - K (4, 10) - D (3, 10) - A (4, 15) - I (5, 12)

*****************************************************************

Algorithme glouton
Value : 169,291891115388 - Weight : 27,981884129337
Loaded : 91 (0,223611267387686, 19,1683501932623) -
26 (0,285179768821774, 7,855
73798597592) - 25 (1,82267926718233, 19,5876727996337) -
13 (1,79861418055306, 1
7,0762902298366) - 72 (2,48491597477576, 19,7482354937812) -
20 (0,9142749621133
67, 7,01825728966773) - 76 (1,56728671470996, 10,5517963648549) -
46 (2,99167926
096901, 12,4924042506574) - 94 (4,36860293819038,
17,2877806784994) - 65 (5,5834
0014218511, 18,8808418525759) - 6 (5,94163965244854,
19,6245239766429)

Descente de gradient
Value : 170,729793808763 - Weight : 29,404489653839
Loaded : 46 (2,99167926096901, 12,4924042506574) -
91 (0,223611267387686, 19,168
3501932623) - 20 (0,914274962113367, 7,01825728966773) -
26 (0,285179768821774,
7,85573798597592) - 25 (1,82267926718233, 19,5876727996337) -
13 (1,798614180553
06, 17,0762902298366) - 88 (4,09834573701878, 11,5954621655845) -
```

```
76 (1,56728671
470996, 10,5517963648549) - 72 (2,48491597477576,
19,7482354937812) - 12 (2,9076
5992966837, 8,72328238036636) - 6 (5,94163965244854,
19,6245239766429) - 94 (4,3
6860293819038, 17,2877806784994)

Recherche tabou
Value : 170,729793808763 - Weight : 29,404489653839
Loaded : 25 (1,82267926718233, 19,5876727996337) -
20 (0,914274962113367, 7,0182
5728966773) - 91 (0,223611267387686, 19,1683501932623) -
13 (1,79861418055306, 1
7,0762902298366) - 88 (4,09834573701878, 11,5954621655845) -
12 (2,9076599296683
7, 8,72328238036636) - 26 (0,285179768821774, 7,85573798597592) -
72 (2,48491597
477576, 19,7482354937812) - 6 (5,94163965244854, 19,6245239766429) -
76 (1,56728
671470996, 10,5517963648549) - 94 (4,36860293819038, 17,2877806784994) -
46 (2,9
9167926096901, 12,4924042506574)

Recuit simulé
Value : 166,681007773933 - Weight : 29,0835969145799
Loaded : 26 (0,285179768821774, 7,85573798597592) -
91 (0,223611267387686, 19,16
83501932623) - 76 (1,56728671470996, 10,5517963648549) -
65 (5,58340014218511, 1
8,8808418525759) - 13 (1,79861418055306, 17,0762902298366) -
72 (2,4849159747757
6, 19,7482354937812) - 25 (1,82267926718233, 19,5876727996337) -
6 (5,9416396524
4854, 19,6245239766429) - 12 (2,90765992966837, 8,72328238036636) -
20 (0,914274
962113367, 7,01825728966773) - 46 (2,99167926096901, 12,4924042506574) -
52 (2,5
6265579376493, 5,95361495667771)

Optimisation par essaim particulaire
Value : 170,729793808763 - Weight : 29,404489653839
Loaded : 20 (0,914274962113367, 7,01825728966773) -
6 (5,94163965244854, 19,6245
239766429) - 25 (1,82267926718233, 19,5876727996337) -
13 (1,79861418055306, 17,
0762902298366) - 72 (2,48491597477576, 19,7482354937812) -
6 (0,285179768821774
, 7,85573798597592) - 91 (0,223611267387686, 19,1683501932623) -
```

```
88 (4,098345737
01878, 11,5954621655845) - 12 (2,90765992966837, 8,72328238036636) -
94 (4,36860
293819038, 17,2877806784994) - 76 (1,56728671470996, 10,5517963648549) -
46 (2,9
9167926096901, 12,4924042506574)

FIN
```

Les métaheuristiques étant pour la plupart aléatoires, tout comme le deuxième problème à résoudre, il est important d'analyser un peu les résultats obtenus.

Sur le premier problème et sur tous les tests effectués, les cinq algorithmes ont toujours trouvé l'optimum de 54. Ce problème reste assez simple, il n'est donc pas très significatif.

Pour la suite, 20 tests ont été effectués sur le deuxième problème. À trois reprises (soit quand même 15 % des cas), tous les algorithmes ont trouvé le même optimum (que l'on peut supposer global, même si rien ne permet de l'affirmer).

Dans les autres cas, un ou plusieurs algorithmes se détachent avec des résultats meilleurs. Au total, l'**algorithme glouton** a trouvé la plus grande valeur dans 50 % des cas.

En effet, celui-ci peut être piégé assez couramment. Imaginons un sac à dos déjà rempli à 16 kg, pour un poids maximum de 20 kg. Si on a alors le choix entre deux boîtes de 3 kg et de 4 kg, avec respectivement une valeur de 12 et de 13, l'algorithme glouton chargera la première boîte (ayant un rapport de 12/3 = 4 points de valeur par kg) au lieu de la deuxième qui a un rapport de 13/4 = 3.25. Le sac à dos est alors chargé à 19 kg, et le dernier kilogramme libre est "perdu". La deuxième boîte, bien que moins prometteuse, aurait pu remplir cet espace et apporter une valeur supérieure.

L'algorithme glouton est donc ici piégé dans au moins la moitié des cas.

L'algorithme par **descente de gradient** ne trouve l'optimum que dans 45 % des cas. En effet, à chaque fois, un seul point de départ est utilisé, et cet algorithme tombe souvent dans des optimums uniquement locaux.

Les trois autres algorithmes (**Recherche tabou**, **Recuit simulé** et **Optimisation par essaims particulaires**) réussissent les mieux, avec respectivement 65 %, 55 % et 60 %. S'ils sont difficiles à départager, ils sont cependant meilleurs que les deux autres algorithmes. En effet, ils sont souvent créés pour pallier les défauts de la descente de gradient, en essayant d'éviter un maximum de rester bloqué dans un optimum local.

En fonction des problèmes, et en jouant sur les différents paramètres, l'un de ces algorithmes peut dépasser les autres, mais sur l'ensemble des problèmes existants, aucun n'est meilleur. En anglais, on appelle ce phénomène le "No Free Lunch", indiquant qu'il n'existe jamais d'algorithme miracle pouvant résoudre tous les problèmes mieux que les autres.

11. Synthèse

Ce chapitre a présenté cinq algorithmes classés comme métaheuristiques. Ceux-ci ont tous pour but de trouver l'optimum d'un problème. S'il est préférable de trouver l'optimum global, lorsqu'il n'existe aucune méthode mathématique pour le faire et que tester toutes les solutions serait trop long, ils sont l'alternative idéale, en permettant de trouver de bons optimums locaux, voire l'optimum global.

Le premier est l'**algorithme glouton**. Il consiste à construire de manière incrémentale une seule solution, en suivant ce qui semble être le plus adéquat pour atteindre l'optimum.

La **descente de gradient** part d'une solution aléatoire. À chaque itération, on regarde le voisinage de celle-ci, et on suit la direction la plus prometteuse. Lorsque plus aucune amélioration n'est disponible dans le voisinage, c'est que l'optimum local a été atteint. Cet algorithme est simple et fonctionne bien, mais il est souvent bloqué dans des optimums locaux et ne trouve donc pas forcément l'optimum global.

La **recherche tabou** a été créée pour permettre d'améliorer la recherche de gradient. En effet, au lieu de se déplacer vers la meilleure position voisine si elle améliore les résultats, on se déplace vers le meilleur voisin, qu'il soit meilleur ou non. Pour éviter de retrouver régulièrement les mêmes positions, on enregistre aussi les dernières positions parcourues, qui sont notées comme étant taboues, et qui ne peuvent donc plus être utilisées. Cette recherche peut donc parcourir plusieurs optimums locaux pour maximiser les chances de trouver l'optimum global.

Le **recuit simulé** est lui inspiré d'un phénomène physique utilisé en métallurgie. Lorsque l'on lance l'algorithme, la probabilité d'accepter de se déplacer vers une solution voisine moins adaptée est forte. Au cours du temps, avec la baisse de la température, cette probabilité va diminuer jusqu'à être nulle. On commence donc par parcourir plusieurs zones de l'ensemble de recherche, puis on va petit à petit se stabiliser vers une zone qui paraît plus intéressante, jusqu'à trouver un optimum, si possible global.

La dernière métaheuristique est l'**optimisation par essaims particulaires**. À la place de faire évoluer une seule solution en regardant à chaque itération toutes ses voisines, on va faire évoluer dans l'environnement une population de solutions. À chaque itération, chaque solution va s'adapter pour se diriger vers les zones présentant le plus d'intérêt. De cette façon, davantage d'espace est parcouru, et les chances de trouver l'optimum global sont donc augmentées.

Sur le problème aléatoire du sac à dos, les trois dernières métaheuristiques sont équivalentes, et ont donné les meilleurs résultats. La difficulté consiste par contre à choisir les bons paramètres pour chaque algorithme.

Chapitre 6
Systèmes multi-agents

1. Présentation du chapitre

Ce chapitre présente les systèmes multi-agents, qui permettent de répondre à une grande variété de problématiques. Dans ces systèmes, plusieurs agents, aux comportements individuels simples, vont travailler de concert pour résoudre des problèmes beaucoup plus complexes.

Ces algorithmes sont inspirés des observations faites en biologie (et plus particulièrement en éthologie). Des colonies d'insectes arrivent à résoudre des problèmes complexes (comme créer une termitière) alors que chaque insecte individuellement n'a pas de grandes capacités. Ce chapitre commence donc par présenter les principales caractéristiques de ces insectes sociaux.

Les caractéristiques minimales d'un système pour qu'il puisse être considéré comme étant multi-agents sont ensuite présentées, ainsi que les différentes catégories d'agents.

Certains algorithmes sont particuliers et sont un champ d'études à eux seuls. Le chapitre continue donc par la présentation de ceux-ci : algorithmes de meutes, colonies de fourmis, systèmes immunitaires artificiels et automates cellulaires.

Plusieurs exemples d'implémentations en C# sont aussi proposés. Enfin, ce chapitre se termine par une synthèse.

2. Origine biologique

Les insectes ont intéressé les chercheurs en biologie et éthologie (l'étude des comportements) depuis longtemps. Même si tous les comportements sociaux ne sont pas encore connus, les grands principes ont été mis à jour.

La majorité de ceux-ci (environ 90 % des espèces) sont solitaires. Chaque insecte vit de son côté, avec peu voire pas de liens avec ses voisins. C'est le cas par exemple de la majorité des araignées, des moustiques, des mouches... Les contacts se limitent à la recherche de nourriture (par exemple pour les mouches), aux zones de vie (comme les larves de moustiques présentes en grand nombre dans les eaux stagnantes) ou aux périodes de reproduction.

Il existe aussi des insectes sociaux, allant de sociétés très simples à des sociétés très complexes et très organisées. Ceux présentant la plus grande organisation sont dits **insectes eusociaux**. Ils présentent les caractéristiques suivantes :

- La population est divisée en **castes**, chaque caste ayant un rôle précis.
- La reproduction est limitée à une caste particulière.
- Les larves et les jeunes sont élevés ensemble au sein de la colonie.

Les insectes eusociaux ne représentent que 2 % des espèces existantes, mais en masse, ils représentent 75 % de l'ensemble des insectes ! Cela prouve que ces sociétés permettent de faire vivre de très nombreux individus.

Les trois espèces eusociales les plus connues sont les abeilles, les termites et les fourmis.

2.1 Les abeilles et la danse

La majorité des espèces d'abeilles sont solitaires. Cependant, l'espèce Apis Mellifera (l'abeille des ruches qui produit le miel) est une espèce eusociale.

Chaque ruche est une société complète. On y retrouve une reine, chargée de pondre les œufs (elle est fertilisée avant de fonder sa colonie et vit jusqu'à quatre ans), des ouvrières (femelles stériles) et quelques mâles (appelés faux-bourdons) qui ne servent qu'à s'accoupler avec les futures reines. Ils meurent d'ailleurs juste après la reproduction.

Les ouvrières vont avoir différents rôles : le soin au couvain (là où se trouvent les larves), l'entretien de la ruche, la recherche de nourriture, la récolte de celle-ci, la défense de la ruche...

Le phénomène le plus captivant chez ces abeilles est leur **communication**. Elles utilisent des phéromones (des substances chimiques odorantes) pour certains comportements (comme le rappel des abeilles s'étant éloignées après qu'un danger ayant dérangé la ruche soit passé). Mais surtout elles utilisent plusieurs **danses**.

Les éclaireuses ont pour mission de trouver de nouvelles sources de nourritures (de pollen), des points d'eau ou des zones de récolte de résine, voire même de nouvelles zones pour créer ou déplacer la ruche. Lorsqu'elles reviennent au nid, elles vont danser pour indiquer aux suiveuses où aller, et ce qu'elles pourront trouver.

Si la source est proche, elles pratiquent la danse en rond, en tournant dans un sens puis dans l'autre. Les suiveuses vont toucher la danseuse pour savoir ce qui a été trouvé par le goût et l'odeur. Si la source est plus loin, à partir d'une dizaine de mètres et jusqu'à plusieurs kilomètres, l'éclaireuse va pratiquer une danse en 8 : l'axe central du 8 par rapport à la verticale indique la direction de la source par rapport au soleil, la taille et la fréquence du frétillement indiquent la distance et l'intérêt de la source. Enfin, la palpation permet là encore aux suiveuses de découvrir la qualité et la substance trouvée.

Cette danse est très précise et permet de localiser la nourriture avec une erreur de moins de 3° sur la direction à prendre. De plus, elles s'adaptent au déplacement du soleil dans le ciel et à la présence du vent qui augmente ou ralentit les temps de trajet.

Grâce à cette communication, chaque suiveuse sait exactement où aller pour trouver de la nourriture.

Chaque abeille a donc des moyens limités, et un rôle très simple, mais la présence des différents rôles et la communication poussée entre les individus permet aux ruches de survivre et de grandir.

2.2 Les termites et le génie civil

Les termites sont aussi des animaux eusociaux. Ils vivent dans d'immenses colonies de plusieurs milliers d'individus, avec la présence de castes.

Au centre de la colonie vivent le roi et la reine, voire des reines secondaires. Ils sont entourés d'ouvriers, de soldats (qui ont des mandibules puissantes, ou pour certaines espèces, la possibilité de lancer des produits chimiques), de jeunes, de larves... Les ouvrières s'occupent entre autres de la nourriture, du soin aux larves, de la construction de la **termitière**.

Celle-ci présente d'ailleurs des caractéristiques impressionnantes : l'intérieur de la termitière a une température et une humidité constantes, et ce malgré les températures extrêmes et variables présentes en Afrique. Ce contrôle du climat à l'intérieur est dû à sa structure très particulière, avec des puits, une cheminée, des piliers, un nid central surélevé... Il y a donc une ventilation passive. De plus, les termitières peuvent atteindre jusqu'à 8 mètres de haut, et une circonférence à la base de 30 mètres.

Les études ont donc cherché comment les termites pouvaient construire de telles structures, appelées **termitières cathédrales** (par leur ressemblance avec nos cathédrales). En réalité, les termites n'ont aucune conscience de la structure globale et des plans à suivre. Chaque termite construit une boulette de terre, et la dépose à un autre endroit avec une probabilité proportionnelle à la quantité de boulettes déjà présentes.

La structure complète est donc **émergente**. Plusieurs agents très simples (les termites) peuvent ainsi résoudre des problèmes complexes (garder un nid à température et hygrométrie constantes).

Il faut savoir que les architectes se sont inspirés des plans de ces termitières pour construire des bâtiments nécessitant peu voire pas d'énergie pour maintenir une température agréable.

2.3 Les fourmis et l'optimisation de chemins

Les fourmis sont aussi des insectes eusociaux. Leurs colonies peuvent comporter jusqu'au million d'individus. Une ou plusieurs reines pondent les œufs. Les mâles, eux, ne servent qu'à la reproduction et meurent après. Les autres fourmis, des femelles stériles, sont les ouvrières.

Les ouvrières ont plusieurs rôles possibles : elles s'occupent des larves, de la fourmilière, de la recherche de nourriture, de la récolte, de la défense de la colonie (les soldats). Certaines espèces de fourmis ont des rôles encore plus poussés : certaines vont attaquer d'autres espèces pour voler les larves et en faire des esclaves, d'autres sont capables d'élever et de garder des troupeaux de pucerons qu'elles traient pour récupérer le miellat.

Toutes ces colonies survivent grâce à la communication entre les membres. Celle-ci se fait via les **phéromones**, captées par les antennes. La principale communication est celle permettant d'indiquer les sources de nourriture. Les fourmis éclaireuses se déplacent aléatoirement. Si l'une d'entre elles trouve de la nourriture, elle rentre au nid avec. Sur le chemin du retour, elle pose des phéromones, dont l'intensité dépend de la nourriture et de la longueur du chemin.

Les autres éclaireuses, lorsqu'elles rencontrent une piste de phéromones, ont tendance à la suivre. La probabilité de les suivre dépend en effet de la quantité de phéromones posée. De plus, les phéromones s'évaporent naturellement.

Grâce à ces règles simples, les fourmis sont capables de déterminer les chemins les plus courts entre le nid et une source de nourriture. En effet, les chemins les plus longs sont moins utilisés que les chemins les plus courts dans le même temps, ce qui renforce ces derniers.

La communication grâce à des modifications de l'environnement (les traces de phéromones) s'appelle la **stigmergie**.

2.4 Intelligence sociale

Toutes ces espèces font donc preuve d'**intelligence sociale**. Chaque individu n'a pas conscience de tout ce qui se joue dans la colonie, mais il accomplit le travail pour lequel il est fait. En l'absence de hiérarchie, des comportements plus complexes apparaissent, comme la construction des termitières. Plus les individus sont nombreux et leurs liens importants, plus le résultat paraît impressionnant.

C'est cette intelligence sociale, permettant de résoudre des problèmes complexes à partir d'individus aux comportements simples, qui a donné naissance aux systèmes multi-agents.

3. Systèmes multi-agents

Toutes les techniques classées comme **systèmes multi-agents** ont pour but de mettre en œuvre cette intelligence sociale, qui est appelée en informatique **intelligence distribuée**. Pour cela, on retrouve :

- Un environnement.
- Des objets fixes ou non, étant des obstacles ou des points d'intérêt.
- Des agents, aux comportements simples.

Le but de l'algorithme n'est jamais réellement codé, la solution va **émerger** de l'interaction de tous ces éléments entre eux.

3.1 L'environnement

Les objets et les agents se trouvent dans un **environnement**. Celui-ci peut être plus ou moins complexe : il peut s'agir d'un espace délimité (comme un hangar ou une forêt), d'un graphe, ou même d'un espace purement virtuel.

L'environnement correspond donc principalement au problème que l'on cherche à résoudre.

Cet environnement doit pouvoir évoluer au cours du temps : les agents peuvent s'y déplacer, ou les objets être modifiés.

3.2 Les objets

L'environnement possède des **objets** avec lesquels les agents peuvent interagir. Ceux-ci peuvent correspondre à des sources de nourriture, des briques de construction, des villes à visiter, des obstacles...

Rien n'impose que ces objets soient ou non transportables, qu'ils soient temporaires ou permanents. Là encore, il faut s'adapter au problème.

Dans certains systèmes, il n'y a aucun objet, seulement des agents. Leur présence est donc facultative.

3.3 Les agents

Les **agents** vont vivre dans l'environnement et interagir avec les objets et agents. Il est nécessaire de définir, en plus du comportement des agents, les relations que ceux-ci ont entre eux. Il peut s'agir de relation hiérarchique ou de liens de communication.

De plus, il est important que chaque agent ait un ensemble d'opérations possibles, à la fois sur les objets (comme les prendre, les transporter ou les utiliser) et sur les autres agents (ils peuvent directement interagir par exemple en échangeant des objets).

Par ces échanges, l'environnement sera modifié, ce qui implique une modification de l'action des agents, jusqu'à ce que la solution (ou une solution considérée comme suffisamment bonne) soit découverte.

4. Classification des agents

Les agents peuvent être de types très différents, en fonction de certaines caractéristiques.

4.1 Perception du monde

La première différence se situe sur la perception du monde qu'auront les agents. Ils peuvent avoir une vue d'ensemble de tout le monde (**perception totale**), ou seulement de ce qui se trouve dans leur voisinage (**perception localisée**). De plus, ils peuvent avoir une **vision symbolique**, ou seulement celle **issue de la perception**.

Par exemple, si on prend des robots qui se déplacent dans un étage d'immeuble, ils peuvent ne connaître que ce qu'ils voient (la pièce qui les entoure) ou avoir une carte préenregistrée de tout l'étage avec leur position dessus (par exemple via un GPS).

De plus, lorsqu'ils ont une caméra, ils peuvent avoir des algorithmes de reconnaissance d'images leur permettant de reconnaître certains objets (loquets des portes, boutons, cibles...) ou simplement devoir agir en fonction des perceptions brutes (l'image telle qu'elle est obtenue par la caméra).

Les chercheurs en systèmes multi-agents ont une préférence pour des perceptions localisées, et ils sont très partagés entre une approche symbolique et une approche purement basée sur les perceptions.

4.2 Prise des décisions

Les agents doivent choisir quoi faire à tout moment. Ils peuvent avoir un plan déterminé d'avance, ce qui est rarement conseillé, ou agir au fur et à mesure en fonction de leurs perceptions.

Ils ont donc une ou plusieurs règles à appliquer. Là encore, il peut s'agir d'un ensemble complexe composé d'un système expert, d'une utilisation de logique floue, d'états et de transitions... Le choix va dépendre du but visé.

Cependant la complexité du problème à résoudre n'a pas de lien avec la complexité des décisions des agents. En effet, avec quelques règles très simples, il est possible de résoudre des problèmes très complexes. Il s'agit donc surtout de trouver le bon système.

Un individu qui agit en réaction directe à ses perceptions est dit **agent réactif**. Au contraire, s'il agit après avoir réfléchi à une décision en fonction de connaissances, il est dit **agent cognitif**.

Si l'on reprend l'exemple des robots qui se déplacent dans un environnement, un robot qui évite les obstacles en allant dans la direction opposée à la détection d'un élément via un capteur de distance par exemple est un agent purement réactif. S'il choisit son trajet pour aller chercher un objet qu'il sait où trouver (par exemple pour aller chercher l'objet demandé par un humain), alors il s'agit d'un agent cognitif qui va choisir la meilleure stratégie et la meilleure trajectoire.

Dans la majorité des cas, on rajoute aussi un **aspect stochastique** : la présence d'un peu d'aléatoire va permettre de rendre le système souvent plus fluide et plus souple.

4.3 Coopération et communication

Les agents ne sont pas seuls. Il est donc important de savoir comment ils vont coopérer ou communiquer.

On peut imaginer des systèmes purement réactifs qui n'échangeraient pas du tout avec leur environnement. On observe alors des comportements de groupe par **émergence**.

Les individus peuvent aussi **communiquer directement**, via des échanges radio, des sons ou des messages visuels (comme des lumières ou des signes). Ces communications peuvent avoir une portée limitée ou non.

Enfin, ils peuvent, comme les fourmis, laisser des traces dans l'environnement et utiliser la **stigmergie** pour communiquer de manière asynchrone. Un exemple de stigmergie est celui du petit poucet qui laisse des petits cailloux blancs pour retrouver son chemin : en modifiant leur environnement, ses frères et lui peuvent rentrer chez eux.

Remarque

De nombreux animaux utilisent la stigmergie, et pas que dans le cas des espèces eusociales. En effet, les animaux qui marquent leur territoire (comme les ours avec les griffures sur les arbres, ou les chevaux qui font des tas de crottins dans leur territoire) utilisent la stigmergie pour dire aux autres individus de ne pas s'approcher et que ce territoire est déjà pris.

Enfin, dans certains cas, les agents peuvent être amenés à **négocier** entre eux pour trouver un consensus qui sera la solution retenue. Dans ce cas, il faut prévoir différents comportements et la possibilité de mener à bien ces négociations.

4.4 Capacités de l'agent

Le dernier point concerne les **capacités** de l'agent. En effet, celui-ci peut avoir de très faibles capacités, ou une grande gamme d'actions possibles.

Les agents réactifs n'ont en général que quelques actions possibles (par exemple tourner à droite ou à gauche). Les agents cognitifs ont souvent une plus grande gamme d'actions (par exemple choisir un chemin, éviter un obstacle...).

De plus, les agents peuvent tous avoir les mêmes capacités ou être spécifiques à une tâche particulière, organisés en castes comme chez les insectes.

Enfin, en fonction des systèmes mis en place, les différents agents peuvent apprendre au cours du temps ou avoir des connaissances figées. Lorsqu'on ajoute de l'**apprentissage**, il faut déterminer l'algorithme sous-jacent (réseau de neurones, algorithme génétique...).

Il existe donc de nombreux types d'agents et c'est à chaque problème qu'il faudra choisir les meilleurs paramètres pour optimiser le résultat obtenu. Cependant, pour un même problème, il existe souvent plusieurs bonnes solutions, ce qui permet une plus grande souplesse dans la mise en place.

5. Principaux algorithmes

Il existe quelques algorithmes particuliers plus connus que les autres, et présentant des environnements, objets et agents définis. Quatre sont ici présentés : les algorithmes de meutes, l'optimisation par colonies de fourmis, les systèmes immunitaires artificiels et les automates cellulaires.

Dans le chapitre sur les métaheuristiques, l'algorithme d'optimisation par essaims particulaires alors présenté pouvait déjà être vu comme un système multi-agents dans lequel chaque solution a une vue globale de toutes les autres solutions et une mémoire (la meilleure solution trouvée jusqu'alors). Ils se déplacent dans l'espace de solution qui sert d'environnement. Il n'y a pas d'objets. Cependant, cette technique est éloignée de la notion d'agents réactifs.

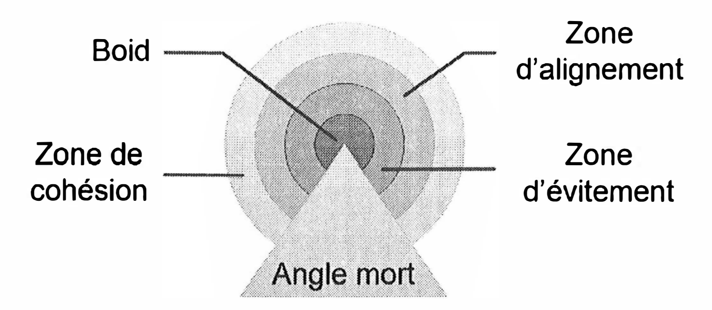
5.1 Algorithmes de meutes

À partir de quelques règles simples, il est possible de simuler des **comportements de meutes** ou de groupes. Craig Reynolds crée ainsi en 1986 les boids, des créatures artificielles évoluant en groupe.

Pour cela, les créatures ont trois comportements, liés à la présence d'autres individus dans leur proximité :

- Un individu très proche va provoquer un comportement d'évitement (pour éviter de rentrer dans un autre individu) : c'est le **comportement de séparation**.
- Un individu proche modifie la direction de la créature, celle-ci ayant tendance à s'aligner sur la direction de son voisin : c'est le **comportement d'alignement**.
- Un individu à distance moyenne va provoquer un rapprochement. En effet, si une créature en voit une autre, elle ira vers elle : c'est le **comportement de cohésion**.

Il est aussi possible d'ajouter un "angle mort" à l'arrière du boid simulant le fait qu'il ne peut pas voir derrière lui.



En fonction des paramètres, en particulier le choix des distances, on peut observer des individus complètement isolés ou au contraire des individus se déplaçant en meutes et pouvant se retrouver après un obstacle, à la manière des bancs de poissons ou des nuées d'oiseaux ou d'insectes.

On observe donc bien une structure émergente à partir de quelques règles très simples. De plus, les trajectoires semblent aléatoires : en réalité l'ensemble est devenu un système complexe, bien que complètement déterministe, qui fait que la moindre modification de l'espace ou du placement initial amène à des mouvements très différents.

■Remarque

On peut alors se servir de ces boids pour représenter des troupeaux en mouvement, en particulier dans les films et les jeux vidéo. L'effet produit est très réaliste (et aussi très hypnotisant en simulation).

5.2 Optimisation par colonie de fourmis

L'**optimisation par colonie de fourmis** est directement inspirée du fonctionnement des fourmis éclaireuses. Le but est donc de trouver une solution optimale grâce à la **stigmergie**. Cette technique a été créée par Marco Dorigo en 1992.

Au départ, l'environnement est vierge. Les fourmis virtuelles vont parcourir l'espace aléatoirement, jusqu'à trouver une solution. La fourmi va alors revenir à son point de départ en déposant des **phéromones**. Les autres agents vont être influencés par ces phéromones et vont avoir tendance à suivre le même chemin.

Au bout de plusieurs itérations, toutes les fourmis suivront le même chemin (représentant la même solution), et l'algorithme aura alors convergé.

L'environnement doit juste représenter, sous la forme d'un graphe ou d'une carte, l'ensemble des solutions. On peut ainsi résoudre les problèmes de recherche de chemins (comme le A*).

Le pseudo-code est le suivant :

```
Initialiser l'environnement
Tant que (critère d'arrêt non atteint)
    Pour chaque fourmi
           Si (cible non atteinte)
                   Se déplacer aléatoirement (en suivant les pistes)
           Sinon
                   Rentrer au nid en laissant des phéromones
           Fin Si
    Fin Pour
    Mettre à jour les traces de phéromones
Fin Tant que
```

La probabilité de suivre une direction dépend de plusieurs critères :

- Les directions possibles.
- La direction d'où vient la fourmi (pour qu'elle ne fasse pas demi-tour).
- Les pistes de phéromones autour.
- D'autres métaheuristiques (pour favoriser par exemple une direction de recherche).

En effet, il faut que la probabilité de suivre une piste augmente avec la quantité de phéromones, sans jamais être à 1 pour ne pas imposer de chemin.

Les phéromones déposées doivent, quant à elles, être proportionnelles à la qualité de la solution ou à sa longueur. Elles peuvent aussi être déposées à taux constant pour certains problèmes.

Celles-ci doivent s'évaporer. Pour cela, à chaque pas de temps, on peut multiplier la quantité par un **taux d'évaporation** inférieur à 1, ou enlever une certaine quantité de phéromones.

Les difficultés se situent principalement dans le choix des probabilités et du taux d'évaporation. En fonction de ces taux, les fourmis peuvent converger trop rapidement vers une solution non optimale ou au contraire ne jamais arriver à converger vers une bonne solution.

5.3 Systèmes immunitaires artificiels

Ces **systèmes immunitaires artificiels** s'inspirent des systèmes naturels des vertébrés (comme le nôtre). En effet, plusieurs cellules collaborent pour déterminer le soi et le non soi, et attaquer ce qui a été déterminé comme étranger (virus, bactéries, champignons, voire poisons).

Les systèmes immunitaires artificiels font donc évoluer, dans un environnement donné, différents agents de défense. Chacun connaît un ensemble de "**menaces**", qu'il sait détecter (et combattre si nécessaire). Ces agents peuvent être créés aléatoirement au départ.

Au cours du temps, si un agent rencontre une menace identifiée, il va l'attaquer, et prévenir les autres agents proches. Ceux-ci vont donc apprendre au contact du premier, et vont à leur tour être capables d'agir contre celle-ci. Les menaces courantes sont donc très vite connues de toute la population d'agents, et la réaction sera rapide.

Cependant, même les menaces plus rares seront détectées car au moins un agent les reconnaîtra. Les agents ont aussi une réaction (bien que plus faible) sur des menaces proches de celles connues. Ils peuvent donc apprendre pour améliorer leurs réponses.

Il est possible de régulièrement injecter des attaques connues pour maintenir la reconnaissance de celles-ci par les agents, comme les vaccins qui sont refaits régulièrement pour entretenir la mémoire de notre propre système immunitaire.

De plus, les agents peuvent se déplacer et **apprendre** les uns des autres, ce qui permet des réponses plus adaptées et plus rapides, même contre des attaques encore inconnues.

Ces systèmes sont particulièrement appréciés en sécurité, pour détecter des fraudes et les combattre (attaques sur les systèmes informatiques, fraudes bancaires...).

5.4 Automates cellulaires

Dans un **automate cellulaire**, l'environnement est une grille régulière. Chaque agent est situé sur une des cases de la grille et ne peut en bouger. Il possède différents états possibles (par exemple des couleurs).

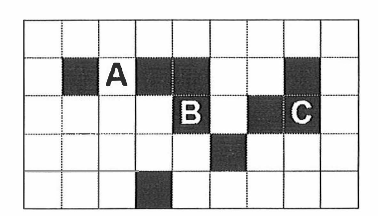
À chaque itération, il va pouvoir changer d'état en suivant quelques règles simples toujours basées sur les états de ses voisins proches. La grille va donc évoluer au cours du temps.

Les comportements émergents des grilles sont complexes, et permettent de créer toutes les formes.

L'automate cellulaire le plus connu est le "**jeu de la vie**" (qui n'a rien d'un jeu vu qu'il n'y a pas d'action possible dessus) :

- Les cellules n'ont que deux états : vivante ou morte (représentés généralement par du noir ou du blanc).
- Elles mettent à jour leur état en fonction de leurs 8 voisines immédiates : toute cellule entourée par exactement trois cellules vivantes devient ou reste vivante, toute cellule vivante entourée d'exactement deux cellules vivantes reste vivante, toutes les autres cellules meurent.

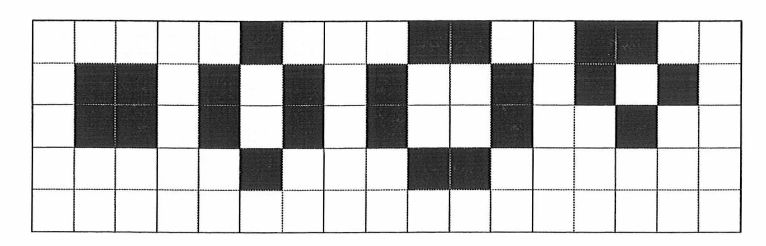
Ainsi, voici une grille avec trois cellules notées A, B et C. La cellule A est morte, les deux autres cellules sont vivantes.



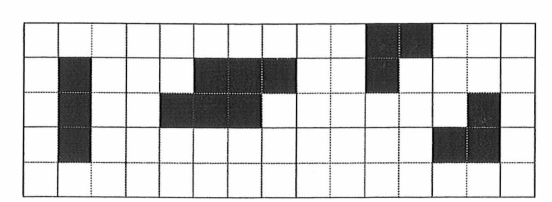
La cellule A est entourée de 2 cellules vivantes seulement, elle reste morte. La cellule B est entourée de 3 cellules vivantes, elle sera donc vivante à la prochaine itération. Enfin, la cellule C est entourée de 2 cellules vivantes, elle conserve donc son état.

Avec ces règles très simples, il est possible d'obtenir des formes stables (qui ne bougent pas au cours du temps), d'autres qui alternent différentes formes en boucle et d'autres qui se déplacent.

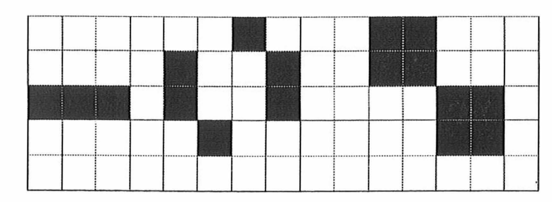
De nombreuses formes ont un nom particulier. Ainsi, voici quatre structures stables : le bloc, la ruche, la mare et le bateau.



Et voici quelques oscillateurs de période 2 : le clignotant, le crapaud et le phare.



Deviennent :



Il a été démontré que toutes les portes logiques (et, ou, non) peuvent être représentées via ce jeu de la vie, ce qui signifie que n'importe quel programme informatique peut être représentés sous la forme d'un dessin dans une grille avec ces quelques règles simples. On peut donc modéliser tout système complexe.

Le jeu de la vie est cependant peu utilisé en pratique mais de nombreux chercheurs travaillent sur ses capacités, en particulier à faire émerger des structures complexes.

6. Domaines d'application

Les **domaines d'application** des systèmes multi-agents sont très nombreux, cela grâce à la diversité des algorithmes.

6.1 Simulation de foules

La première utilisation est la **simulation de foules**. De nombreux logiciels utilisent des agents pour simuler des personnes se déplaçant dans un lieu, permettant ainsi de comprendre les réactions en cas d'évacuation, et découvrir les zones de bousculades potentielles.

On retrouve aussi cette simulation dans le domaine du trafic routier, pour comprendre et simuler les modifications induites par des changements comme l'ajout de feux, ou la réduction de la vitesse sur certaines portions.

Il est aussi possible de simuler des troupes, que ce soient des guerriers, des joueurs, des animaux... Ces simulations sont très utilisées dans le monde du loisir, car elles permettent d'avoir des animations à faible coût.

Ainsi, le logiciel MASSIVE, un des leaders du marché, a servi à créer de nombreuses publicités (comme pour Adidas, Coca Cola, Pepsi...) et surtout de nombreux films, pour simuler les grands mouvements de foule (et limiter les coûts en figurants). Le logiciel a ainsi été utilisé dans les films suivants (entre autres) : La planète des singes, l'affrontement (2014), World War Z (2013), Godzilla (2014), Pompei (2014), Tron Legacy (2010), Le Seigneur des Anneaux (2003), I, Robot (2004)...

Dans les dessins animés ou les jeux vidéo, ces algorithmes sont aussi utilisés pour représenter les foules en arrière-plan (par exemple les spectateurs), sans avoir à les coder personnage par personnage, et en les rendant "vivants".

6.2 Planification

Le deuxième grand domaine d'utilisation est la **planification**. En effet, en particulier via les algorithmes à base de fourmis, il est possible de résoudre tout problème se présentant sous la forme d'un graphe.

On peut ainsi choisir et optimiser l'usage des différentes machines dans une usine en fonction des commandes et des matières premières disponibles, optimiser une flotte de véhicules en les dispatchant plus efficacement ou encore améliorer les horaires d'un service comme le train.

Des applications réelles ont donc vu le jour sur des variantes du problème du voyageur de commerce, pour l'utilisation et l'organisation d'usines, pour la recherche de chemins en s'adaptant à la circulation (pour contourner les bouchons par exemple) ou encore pour le routage de paquets dans des réseaux...

Ces algorithmes sont particulièrement performants dans des environnements dynamiques, car ils permettent une adaptation en temps réel de la solution proposée.

6.3 Phénomènes complexes

Le troisième domaine concerne toute la modélisation et la compréhension de **phénomènes complexes**.

On les retrouve ainsi en biologie pour la simulation de la croissance des populations de bactéries en fonction du milieu (via des automates cellulaires), pour le repliement de protéines, la croissance des plantes...

En physique, ils permettent de simuler des phénomènes complexes en les découpant en petits problèmes plus simples et en laissant l'émergence créer le système entier. Ils peuvent par exemple simuler les gouttes d'eau, le brouillard, les flammes ou encore l'écoulement de liquides.

Enfin, en finance, ils peuvent permettre d'optimiser les portefeuilles d'actions, ou les investissements.

Les systèmes multi-agents peuvent donc être utilisés dans de très nombreux domaines et pour de nombreuses applications. La difficulté réside surtout dans le choix de l'environnement, des agents et surtout de leurs caractéristiques.

7. Implémentation

Plusieurs exemples sont ici implémentés. Par leur fonctionnement, ces algorithmes sont principalement graphiques, aussi les codes présentés ici, s'ils sont génériques pour les classes de base, sont des applications WPF pour Windows.

Lors de la création de tels projets avec Visual Studio, des fichiers App.config, App.xaml et App.xaml.cs sont créés. Ceux-ci ne sont pas modifiés. Par contre, les fichiers MainWindows.xaml et MainWindows.xaml.cs sont, eux, modifiés et le code est donné.

Le modèle MVVM est volontairement non respecté, pour garder le code plus léger, et simplifier sa compréhension.

7.1 Banc de poissons

La première application est une simulation d'un banc de poissons, inspirée des boids de Reynolds, en deux dimensions.

Nous allons donc voir un ensemble de poissons, représentés sous la forme de traits, se déplacer dans un océan virtuel et éviter des zones dangereuses à l'intérieur de celui-ci (cela peut représenter des obstacles physiques ou des zones présentant des prédateurs).

Le comportement du banc sera donc uniquement obtenu par émergence.

7.1.1 Les objets du monde et les zones à éviter

Avant de coder les agents eux-mêmes, nous allons coder une première classe qui peut être utilisée à la fois pour les objets et les agents. Celle-ci, nommée **ObjectInWorld**, présente deux attributs PosX et PosY indiquant les coordonnées de l'objet. Il s'agit de champs publics pour optimiser les accès à ceux-ci. En effet, il y aura de nombreux accès et l'appel d'une méthode (avec la création de son contexte) serait une perte de temps notable.

La base de notre classe est donc la suivante. On a créé deux constructeurs, un par défaut et un initialisant les deux attributs.

```
using System;

public class ObjectInWorld
{
    public double PosX;

    public double PosY;

    public ObjectInWorld() {}

    public ObjectInWorld(double _x, double _y)
    {
        PosX = _x;
        PosY = _y;
    }
}
```

On ajoute une méthode permettant de calculer la distance entre l'objet et un autre objet de l'environnement.

```
    public double DistanceTo(ObjectInWorld _object)
    {
        return Math.Sqrt((_object.PosX - PosX) * (_object.PosX -
PosX) + (_object.PosY - PosY) * (_object.PosY - PosY));
    }
```

Cette distance est souvent demandée, et elle fait appel à une racine carrée. Pour optimiser un peu les algorithmes, on propose donc une deuxième méthode qui calcule la distance au carré.

```
    public double SquareDistanceTo(ObjectInWorld _object)
    {
           return (_object.PosX - PosX) * (_object.PosX - PosX) +
(_object.PosY - PosY) * (_object.PosY - PosY);
    }
```

Les zones à éviter, **BadZone**, sont des objets situés, qui possèdent une propriété indiquant leur portée (Radius), et le temps restant de vie de la zone (car elles devront disparaître automatiquement) noté timeToLive.

Cette classe possède aussi trois méthodes : un constructeur avec paramètres, une méthode Update qui décrémente le temps de vie restant et une méthode Dead qui renvoie vrai si le temps restant est arrivé à 0.

Le code de la classe est donc le suivant :

```
public class BadZone : ObjectInWorld
{
    protected double radius;
    public double Radius
    {
          get
          {
                  return radius;
          }
    }

    protected int timeToLive = 100;

    public BadZone(double _posX, double _posY, double _radius)
    {
          PosX = _posX;
          PosY = _posY;
          radius = _radius;
    }

    public void Update()
    {
          timeToLive--;
    }
```

```
    public bool Dead()
    {
        return timeToLive <= 0;
    }
}
```

7.1.2 Les agents-poissons

On peut alors passer à la classe **FishAgent** représentant nos agents-poissons. Ceux-ci héritent de la classe ObjectInWorld. On leur rajoute tout d'abord plusieurs constantes, qui pourront être modifiées si besoin :

- La distance parcourue à chaque itération (STEP) en unité arbitraire.
- La distance indiquant quelle est la zone d'évitement (DISTANCE_MIN) et sa version au carré (SQUARE_DISTANCE_MIN) pour l'optimisation des calculs.
- La distance indiquant jusqu'où va la zone d'alignement (DISTANCE_MAX) et sa version au carré (SQUARE_DISTANCE_MAX).

De plus, la direction des poissons est représentée par le déplacement en x et le déplacement en y à chaque itération. On les code via des propriétés SpeedX et SpeedY.

On ajoute un constructeur qui prend la position de départ et l'angle pris par le poisson. Le code de base est le suivant :

```
using System;
using System.Collections.Generic;
using System.Linq;

public class FishAgent : ObjectInWorld
{
    protected const double STEP = 3;
    protected const double DISTANCE_MIN = 5;
    protected const double SQUARE_DISTANCE_MIN = 25;
    protected const double DISTANCE_MAX = 40;
    protected const double SQUARE_DISTANCE_MAX = 1600;

    protected double speedX;
    public double SpeedX { get { return speedX; } }
```

```
        protected double speedY;
        public double SpeedY { get { return speedY; } }

        internal FishAgent(double _x, double _y, double _dir)
        {
                PosX = _x;
                PosY = _y;
                speedX = Math.Cos(_dir);
                speedY = Math.Sin(_dir);
        }
}
```

La première méthode que nous ajoutons permet de calculer la nouvelle position du poisson. Il s'agit donc simplement d'ajouter à la position actuelle la vitesse multipliée par la longueur du déplacement :

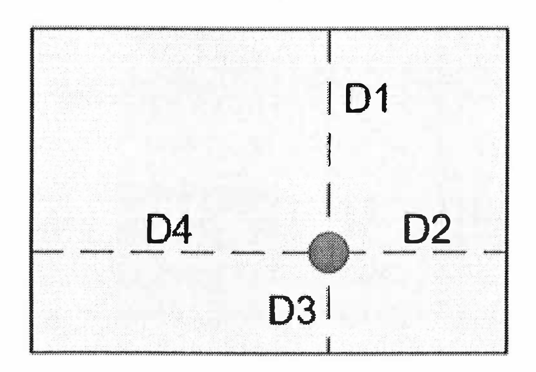
```
        internal void UpdatePosition()
        {
                PosX += STEP * SpeedX;
                PosY += STEP * SpeedY;
        }
```

La méthode suivante permet de savoir si un autre poisson est proche, c'est-à-dire dans la zone d'alignement (donc entre la distance minimale et la distance maximale). Cette méthode utilise les distances au carré.

```
        private bool Near(FishAgent _fish)
        {
                double squareDistance = SquareDistanceTo(_fish);
                return squareDistance < SQUARE_DISTANCE_MAX &&
squareDistance > SQUARE_DISTANCE_MIN;
        }
```

Dans la simulation, les poissons doivent éviter de rentrer dans les autres poissons mais aussi dans les murs. Cependant les murs ne sont pas localisés en un point donné, et il est donc nécessaire de calculer la distance aux murs. `DistanceToWall` renvoie la plus petite distance.

Dans le cas suivant par exemple, la distance renvoyée serait D3.



Le code de cette méthode est donc le suivant :

```
internal double DistanceToWall(double _wallXMin, double
_wallYMin, double _wallXMax, double _wallYMax)
{
        double min = double.MaxValue;
        min = Math.Min(min, PosX - _wallXMin);
        min = Math.Min(min, PosY - _wallYMin);
        min = Math.Min(min, _wallYMax - PosY);
        min = Math.Min(min, _wallXMax - PosX);
        return min;
}
```

Pour simplifier le calcul des vitesses dans les différents cas qui se présentent, nous ajoutons une fonction permettant de normaliser celles-ci. En effet, nous ferons en sorte que la vitesse d'un poisson soit constante dans le temps. On normalise donc le vecteur vitesse :

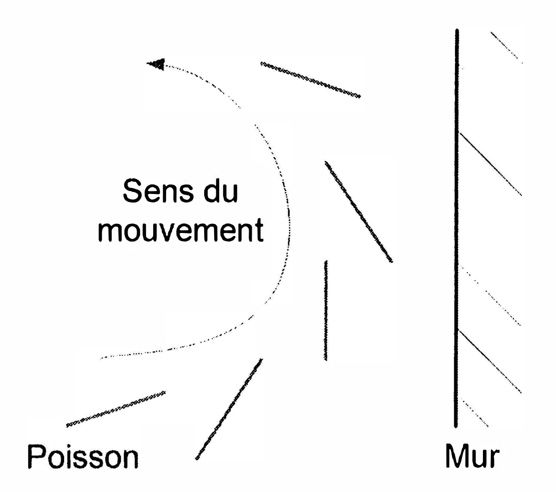
```
protected void Normalize()
{
        double speedLength = Math.Sqrt(SpeedX * SpeedX + SpeedY
* SpeedY);
        speedX /= speedLength;
        speedY /= speedLength;
}
```

Le comportement du poisson est donc simple :

- S'il y a un mur ou une zone à éviter dans la zone très proche, alors on l'évite (règles 1 et 2).
- S'il y a un poisson dans la zone très proche, on s'en éloigne (règle 3).
- S'il y a un poisson dans la zone proche, on s'aligne sur lui (règle 4).

Quatre méthodes sont donc nécessaires, une par comportement. On commence par éviter les murs. Pour cela, il faut tout d'abord s'arrêter au mur si le déplacement aurait permis de sortir de notre océan virtuel. Ensuite, on modifie la direction du poisson en fonction du mur : les murs horizontaux modifient la vitesse horizontale, sans modifier la vitesse verticale, de manière à faire tourner le poisson tout en conservant globalement sa direction actuelle.

Voici le schéma d'un poisson arrivant sur un mur et la trajectoire qu'il aura :



On termine la méthode en normalisant le nouveau vecteur, et la méthode renvoie vrai si on a détecté un mur (car dans ce cas, aucun autre comportement ne peut s'appliquer).

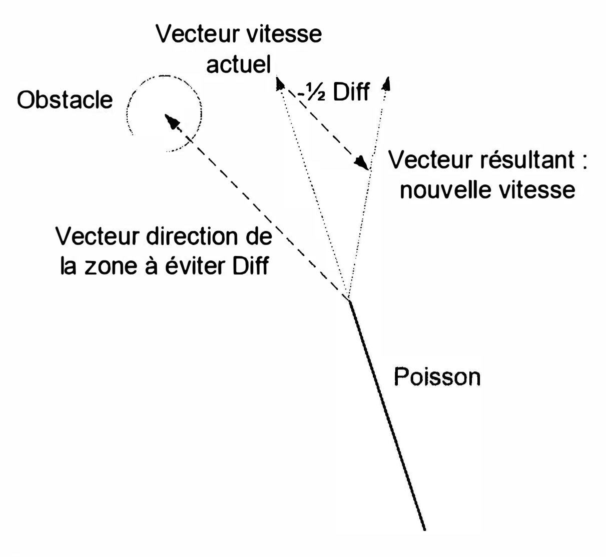
```
   internal bool AvoidWalls(double _wallXMin, double _wallYMin,
double _wallXMax, double _wallYMax)
   {
          // S'arrêter aux murs
          if (PosX < _wallXMin)
```

```
            {
                    PosX = _wallXMin;
            }
            if (PosY < _wallYMin)
            {
                    PosY = _wallYMin;
            }
            if (PosX > _wallXMax)
            {
                    PosX = _wallXMax;
            }
            if (PosY > _wallYMax)
            {
                    PosY = _wallYMax;
            }

            // Changer de direction
            double distance = DistanceToWall(_wallXMin, _wallYMin,
_wallXMax, _wallYMax);
            if (distance < DISTANCE_MIN)
            {
                    if (distance == (PosX - _wallXMin))
                    {
                            speedX += 0.3;
            }
            else if (distance == (PosY - _wallYMin))
            {
                            speedY += 0.3;
            }
            else if (distance == (_wallXMax - PosX))
            {
                            speedX -= 0.3;
            }
            else if (distance == (_wallYMax - PosY))
            {
                            speedY -= 0.3;
            }
            Normalize();
            return true;
            }
            return false;
    }
```

Pour éviter les obstacles, on va chercher la zone à éviter la plus proche de laquelle on est. S'il y a effectivement un obstacle très proche de nous, alors on va calculer le vecteur direction `Diff` entre nous et l'obstacle. On applique une modification de notre vecteur vitesse en y retranchant la moitié de ce vecteur `Diff`. On termine en normalisant le nouveau vecteur direction, et on renvoie vrai si on a dû éviter une zone.

Voici par exemple le résultat obtenu : l'ancien vecteur direction est modifié pour s'écarter de la zone à éviter.



Le code de la fonction d'évitement est donc :

```
    internal bool AvoidObstacle(List<BadZone> _obstacles)
    {
            BadZone nearestObstacle = _obstacles.Where(x =>
SquareDistanceTo(x) < x.Radius*x.Radius).FirstOrDefault();

            if (nearestObstacle != null)
            {
                    double distanceToObstacle = DistanceTo(nearestObstacle);
                    double diffX = (nearestObstacle.PosX - PosX) /
distanceToObstacle;
                    double diffY = (nearestObstacle.PosY - PosY) /
distanceToObstacle;

                    speedX = SpeedX - diffX/2;
```

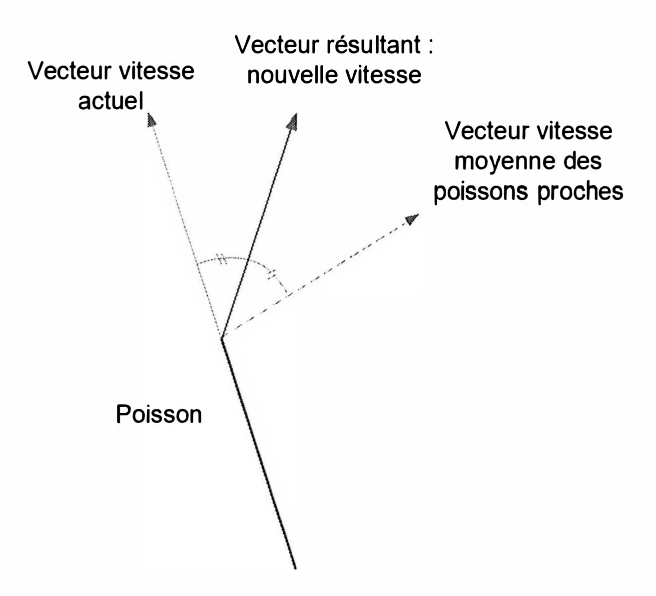
```
                speedY = SpeedY - diffY/2;
                Normalize();
                return true;
            }
            return false;
    }
```

Pour éviter un poisson trop proche de manière souple, on calcule là encore le vecteur unitaire entre l'agent et ce poisson, que l'on retranche à sa propre direction (en réalité, seul le quart de la différence est retranché). Là encore on termine en normalisant notre vecteur vitesse et on renvoie vrai si on a évité un poisson.

```
    internal bool AvoidFish(FishAgent _fishAgent)
    {
            double squareDistanceToFish =
SquareDistanceTo(_fishAgent);
            if (squareDistanceToFish < SQUARE_DISTANCE_MIN)
            {
                    double diffX = (_fishAgent.PosX - PosX) /
Math.Sqrt(squareDistanceToFish);
                    double diffY = (_fishAgent.PosY - PosY) /
Math.Sqrt(squareDistanceToFish);

                    speedX = SpeedX - diffX/4;
                    speedY = SpeedY - diffY/4;
                    Normalize();
                    return true;
            }
            return false;
    }
```

Le dernier comportement est le comportement d'alignement. Dans ce cas-là, on cherche tout d'abord tous les poissons dans notre zone d'alignement. La nouvelle direction du poisson est une moyenne entre la direction des autres poissons et sa direction actuelle, pour là encore avoir une certaine fluidité dans les mouvements.



Pour terminer, le vecteur vitesse est normalisé.

```
    internal void ComputeAverageDirection(FishAgent[] _fishList)
    {
        List<FishAgent> fishUsed = _fishList.Where(x =>
Near(x)).ToList();
        int nbFish = fishUsed.Count;
        if (nbFish >= 1)
        {
            double speedXTotal = 0;
            double speedYTotal = 0;
            foreach (FishAgent neighbour in fishUsed)
            {
                speedXTotal += neighbour.SpeedX;
                speedYTotal += neighbour.SpeedY;
            }

            speedX = (speedXTotal / nbFish + speedX) /2;
            speedY = (speedYTotal / nbFish + speedY) /2;
            Normalize();
        }
    }
```

La dernière méthode est celle permettant de mettre à jour les poissons et qui correspond donc à leur fonctionnement global. Pour cela, on cherche tout d'abord si on doit éviter un mur, puis un obstacle et enfin un poisson. Si on n'a pas eu d'évitement à faire, on passe au comportement d'alignement. Enfin, une fois la nouvelle direction calculée, on calcule la nouvelle position par `UpdatePosition`.

```
    internal void Update(FishAgent[] _fishList, List<BadZone>
_obstacles, double _max_width, double _max_height)
    {
            if (!AvoidWalls(0, 0, _max_width, _max_height))
            {
                    if (!AvoidObstacle(_obstacles))
                    {
                            double squareDistanceMin =
_fishList.Where(x => !x.Equals(this)).Min(x =>
x.SquareDistanceTo(this));
                            if (!AvoidFish(_fishList.Where(x =>
x.SquareDistanceTo(this) == squareDistanceMin).FirstOrDefault()))
                            {
                                    ComputeAverageDirection(_fishList);
                            }
                    }
            }
            UpdatePosition();
    }
```

Les agents sont donc complètement codés.

7.1.3 L'océan

L'environnement du banc de poissons est un océan virtuel. Celui-ci est mis à jour à la demande, de manière asynchrone. Il est donc nécessaire que l'océan puisse lever un évènement lorsque la mise à jour est complète (pour lancer la mise à jour de l'interface graphique).

Nous utilisons un `delegate` pour cela. Le fichier Ocean.cs commence donc par déclarer un `delegate` qui prend en paramètres les poissons et les obstacles. La classe **Ocean** déclare, elle, un évènement du type de notre `delegate`.

De plus, l'océan comporte un tableau de poissons (pour des raisons d'optimisation, comme le nombre de poissons est fixe, on préfère un tableau à une liste) et une liste d'obstacles (qui sont en nombre variable).

La base de l'océan est donc la suivante :

```
using System;
using System.Collections.Generic;
using System.Linq;

public delegate void OceanUpdated(FishAgent[] _fish,
List<BadZone> _obstacles);

public class Ocean
{
    public event OceanUpdated oceanUpdatedEvent;

    FishAgent[] fishList = null;
    List<BadZone> obstacles = null;

    // Suite du code
}
```

Comme les poissons sont créés aléatoirement au départ, l'océan comporte un générateur aléatoire. Il possède aussi deux champs indiquant sa taille (`MAX_WIDTH` pour la largeur et `MAX_HEIGHT` pour la hauteur).

```
    Random randomGenerator;

    protected double MAX_WIDTH;
    protected double MAX_HEIGHT;
```

Cet océan possède ensuite un constructeur. Il va tout d'abord appliquer la taille passée en paramètre, puis initialiser le nombre de poissons demandé en paramètre (chacun est positionné et dirigé aléatoirement). La liste des obstacles est vide à l'origine.

```
    public Ocean(int _fishNb, double _width, double _height)
    {
          MAX_WIDTH = _width;
          MAX_HEIGHT = _height;
          randomGenerator = new Random();

          fishList = new FishAgent[_fishNb];
          obstacles = new List<BadZone>();
          for (int i = 0; i < _fishNb; i++)
          {
                 fishList[i] = new
FishAgent(randomGenerator.NextDouble() * MAX_WIDTH,
randomGenerator.NextDouble() * MAX_HEIGHT,
randomGenerator.NextDouble() * 2 * Math.PI);
          }
    }
```

On pourra depuis l'interface ajouter des obstacles. La méthode `AddObstacle` crée donc une nouvelle zone à éviter aux coordonnées indiquées, avec la portée demandée et l'ajoute à la liste actuelle.

```
    public void AddObstacle(double _posX, double _posY, double
_radius) {
          obstacles.Add(new BadZone(_posX, _posY, _radius));
    }
```

La mise à jour des obstacles consiste simplement à demander à chaque zone de se mettre à jour (c'est-à-dire de baisser son temps restant à vivre) puis de supprimer les zones qui ont atteint leur fin de vie.

```
    private void UpdateObstacles()
    {
          foreach (BadZone obstacle in obstacles)
          {
                obstacle.Update();
          }
          obstacles.RemoveAll(x => x.Dead());
    }
```

On appelle pour chaque poisson sa méthode de mise à jour :

```
    private void UpdateFish()
    {
          foreach (FishAgent fish in fishList)
          {
```

```
                fish.Update(fishList, obstacles, MAX_WIDTH, MAX_HEIGHT);
            }
        }
```

La méthode principale est appelée depuis l'interface. Elle consiste à demander la mise à jour de tout l'océan. On met donc à jour les obstacles puis les poissons. Enfin, on lance l'évènement si des classes écoutent, pour indiquer que cette mise à jour est terminée.

```
    public void UpdateEnvironnement()
    {
        UpdateObstacles();
        UpdateFish();
        if (oceanUpdatedEvent != null)
        {
            oceanUpdatedEvent(fishList, obstacles);
        }
    }
```

Toutes les classes de base sont maintenant implémentées. Il ne reste donc plus qu'à rajouter l'interface.

7.1.4 L'application graphique

Le programme est une application WPF, graphique. La fenêtre principale **MainWindows** possède donc un fichier xaml pour le dessin de l'interface et un fichier xaml.cs pour le code-behind.

Voici le code de notre fenêtre, très simple, car elle ne contient qu'un canvas bleu azur. La fenêtre fait 525 par 350 pixels, et s'appelle "**FishSimulator**".

```
<Window x:Class="MainWindow"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="FishSimulator" Height="350" Width="525">
    <Canvas x:Name="oceanCanvas" Height="Auto" Width="Auto"
Background="Azure"/>
</Window>
```

Le code-behind est plus long. On ne suit pas le modèle MVVM pour simplifier le code.

La fenêtre possède donc une référence vers un océan. Son constructeur se contente d'initialiser ses composants puis s'abonne à l'évènement Loaded indiquant que la fenêtre est chargée :

```
using MultiAgentSystemPCL;
using System;
using System.Collections.Generic;
using System.Windows;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Shapes;
using System.Windows.Threading;

public partial class MainWindow : Window
{
    Ocean myOcean;

    public MainWindow()
    {
           InitializeComponent();

           Loaded += MainWindow_Loaded;
    }

    // Suite du code ici
}
```

La méthode lancée sur l'évènement loaded va s'abonner aux clics de la souris (ce qui nous permettra de rajouter des zones à éviter), créer un océan et l'initialiser, s'abonner à son évènement de fin de mise à jour, et créer un timer qui nous permettra de lancer de manière régulière le même code. Le temps choisi doit être adapté aux capacités de l'ordinateur sur lequel tourne la simulation. Ici, on a choisi un temps de 15ms. Enfin, on lance le timer.

```
    void MainWindow_Loaded(object _sender, RoutedEventArgs _e)
    {
           oceanCanvas.MouseDown += oceanCanvas_MouseDown;

           myOcean = new Ocean(250, oceanCanvas.ActualWidth,
oceanCanvas.ActualHeight);
           myOcean.oceanUpdatedEvent += myOcean_oceanUpdatedEvent;

           DispatcherTimer dispatcherTimer = new
```

```
DispatcherTimer();
            dispatcherTimer.Tick += dispatcherTimer_Tick;
            dispatcherTimer.Interval = new TimeSpan(0, 0, 0, 0, 15);
            dispatcherTimer.Start();
    }
```

La méthode lancée par le `timer` consiste juste à demander la mise à jour de l'océan :

```
    void dispatcherTimer_Tick(object _sender, EventArgs _e)
    {
            myOcean.UpdateEnvironnement();
    }
```

Sur un clic de souris, on rajoute une zone à éviter à l'endroit du clic :

```
     void oceanCanvas_MouseDown(object _sender,
MouseButtonEventArgs _mouseEvent)
     {
     myOcean.AddObstacle(_mouseEvent.GetPosition(oceanCanvas).X,
_mouseEvent.GetPosition(oceanCanvas).Y, 10);
     }
```

Les méthodes restantes sont les méthodes graphiques. Un poisson est représenté par un trait de 10px, partant de la tête du poisson et allant vers sa queue dont les coordonnées sont calculées à partir de la direction du poisson (et donc sa vitesse en x et en y).

```
    private void DrawFish(FishAgent _fish)
    {
          Line body = new Line();
          body.Stroke = Brushes.Black;
          body.X1 = _fish.PosX;
          body.Y1 = _fish.PosY;
          body.X2 = _fish.PosX - 10 * _fish.SpeedX;
          body.Y2 = _fish.PosY - 10 * _fish.SpeedY;
          oceanCanvas.Children.Add(body);
    }
```

Les obstacles sont représentés par des cercles, centrés sur leurs positions, et d'un rayon correspondant à la portée de ceux-ci.

```
    private void DrawObstacle(BadZone _obstacle)
    {
          Ellipse circle = new Ellipse();
          circle.Stroke = Brushes.Black;
          circle.Width = 2 * _obstacle.Radius;
          circle.Height = 2 * _obstacle.Radius;
          circle.Margin = new Thickness(_obstacle.PosX -
_obstacle.Radius, _obstacle.PosY - _obstacle.Radius, 0, 0);
          oceanCanvas.Children.Add(circle);
    }
```

La dernière méthode est celle qui est lancée lorsque l'on reçoit l'évènement de fin de mise à jour de l'océan. Dans ce cas-là, on efface le `canvas`, puis on dessine tous les poissons puis tous les obstacles. On met enfin à jour l'affichage.

```
    void myOcean_oceanUpdatedEvent(FishAgent[] _fish,
List<BadZone> _obstacles)
    {
          oceanCanvas.Children.Clear();

          foreach (FishAgent fish in _fish)
          {
                DrawFish(fish);
          }

          foreach (BadZone obstacle in _obstacles)
          {
                DrawObstacle(obstacle);
          }
          oceanCanvas.UpdateLayout();
    }
```

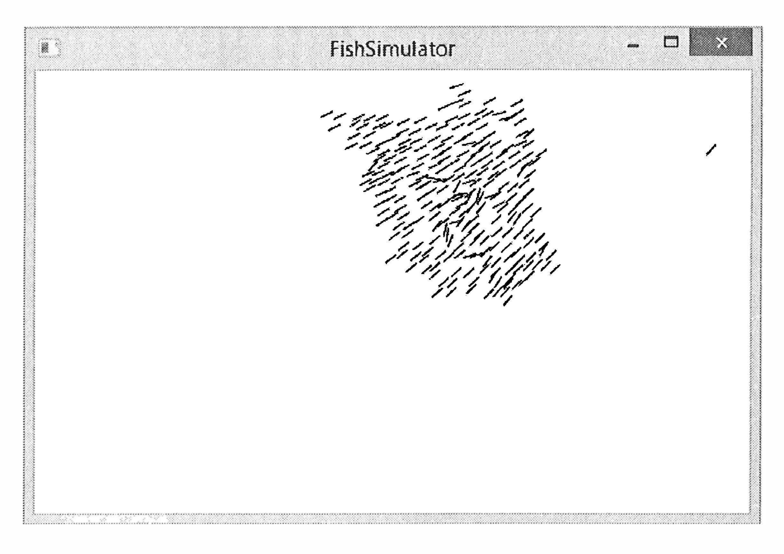
L'application est entièrement fonctionnelle.

7.1.5 Résultats obtenus

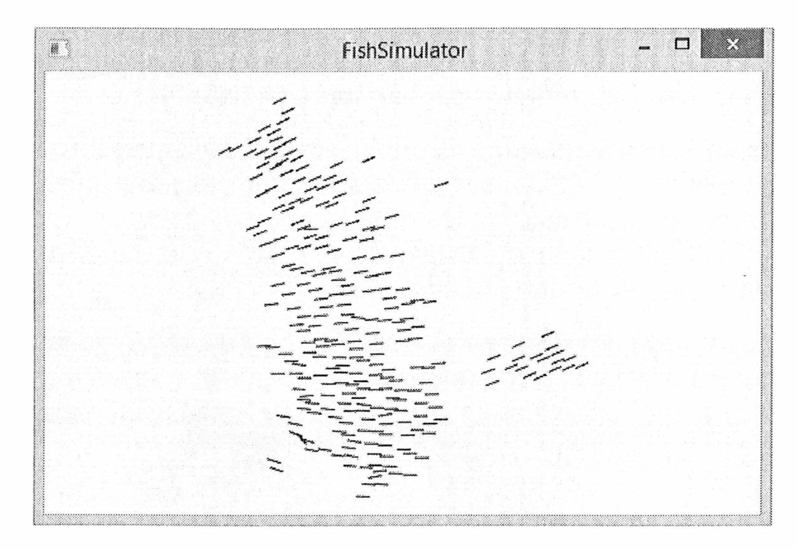
Les poissons se rapprochent très rapidement entre eux, pour créer plusieurs groupes qui finissent par fusionner. Ils se déplacent alors en bancs, et évitent les obstacles. Selon les dispositions, le groupe peut se séparer en deux groupes ou plus, mais le banc total finit toujours par se reformer.

Le comportement en bancs de poissons est donc totalement émergent, les règles codées étant très simples et uniquement basées sur un voisinage proche ou très proche. Contrairement aux boids de Reynolds, nos poissons n'ont même pas de zone de cohérence, ce qui explique que le groupe se scinde parfois, mais pourtant le banc se reconstruit ensuite.

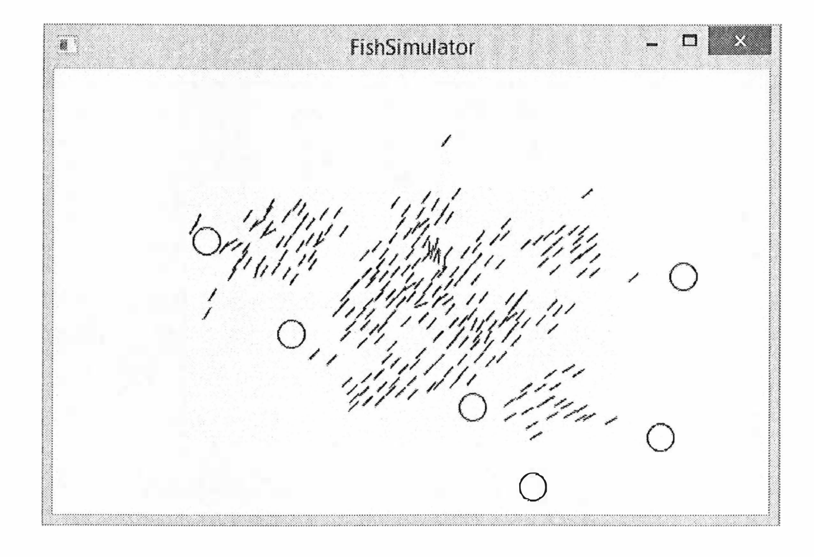
Voici par exemple un banc de poissons compact. On observe qu'un poisson s'est détaché, il rejoindra le banc ultérieurement.



Ici, on observe que le banc se resserre. Un groupe de poissons à droite vient les rejoindre :



Dans cette dernière capture, avec zones à éviter, on voit que le banc s'est temporairement séparé en allant vers le coin en haut à droite. Il se reformera ultérieurement.



7.2 Tri sélectif

La deuxième application propose de faire trier des déchets par des petits robots virtuels. En effet, plusieurs détritus sont déposés dans l'environnement, de trois types possibles. Les différents agents ont juste pour ordre de ramasser les objets pour les déposer là où il y en a au moins un autre du même type.

La probabilité de prendre un objet dépend du nombre d'objets déjà présents (ainsi il est plus rare de prendre un objet dans un tas conséquent). Si les robots passent à proximité d'un tas du même type, alors ils déposent leur charge.

Le tri effectif des déchets en trois tas (un par type) n'est là encore qu'émergence. Aucune communication ou synchronisation entre eux n'est nécessaire. Cette simulation s'inspire des termites et de la construction des termitières cathédrales.

7.2.1 Les déchets

Les déchets sont des objets placés dans le monde, tout comme les agents. Nous réutilisons la classe **ObjectInWorld** créée pour la simulation de bancs de poissons :

```
using System;

public class ObjectInWorld
{
    public double PosX;

    public double PosY;

    public ObjectInWorld() {}

    public ObjectInWorld(double _x, double _y)
    {
          PosX = _x;
          PosY = _y;
    }

    public double DistanceTo(ObjectInWorld _object)
    {
          return Math.Sqrt((_object.PosX - PosX) * (_object.PosX -
PosX) + (_object.PosY - PosY) * (_object.PosY - PosY));
```

```
    }

    public double SquareDistanceTo(ObjectInWorld _object)
    {
        return (_object.PosX - PosX) * (_object.PosX - PosX) +
(_object.PosY - PosY) * (_object.PosY - PosY);
    }
}
```

En plus d'hériter de cette classe, les déchets implémentés par la classe **`Waste`** possèdent deux propriétés :

– Une indiquant le type du déchet, sous la forme d'un entier.

– Une indiquant la taille du tas en nombre de déchets posés à cet endroit.

On rajoute de plus une constante permettant de savoir à quelle vitesse la probabilité de prendre un élément sur un tas diminue avec la taille.

Ici, elle est de 0.6. Cela signifie que si la probabilité de prendre un élément seul est de 100 %, la probabilité de prendre un élément sur une pile de 2 est de 60 %, celle d'en prendre un sur une pile de trois est de 60*0.6 = 36 %, celle d'en prendre un sur une pile de quatre de 36*0.6 = 21.6 % et ainsi de suite.

La base de la classe est donc la suivante :

```
public class Waste : ObjectInWorld
{
    private const double PROBA_DECREASE = 0.6;

    protected int type;
    public int Type
    {
        get { return type; }
    }

    protected int size = 1;
    public int Size
    {
        get { return size; }
    }

    // Suite du code ici
}
```

Deux constructeurs sont disponibles : un utilisant des paramètres sur la position et le type d'un tas, et l'autre par copie d'un élément existant (mais avec une taille initiale de 1). Ce deuxième constructeur sera utile lorsqu'un agent prendra un élément d'un tas.

```
public Waste(double _posX, double _posY, int _type)
{
   PosX = _posX;
   PosY = _posY;
   type = _type;
}

public Waste(Waste _goal)
{
   PosX = _goal.PosX;
   PosY = _goal.PosY;
   type = _goal.type;
}
```

Chaque tas a une zone d'influence représentant sa portée. En effet, plus un tas est gros et plus il va attirer les agents autour (il est ainsi plus visible à la façon des montagnes). Ici, un élément seul a une visibilité de 10, et chaque élément supplémentaire rajoute 8 de visibilité.

```
public int Zone
{
       get { return 10 + (8 * size - 1); }
}
```

On rajoute deux méthodes permettant d'incrémenter ou de décrémenter la taille d'un tas, ce qui représente un agent posant ou prenant un élément.

```
internal void Decrease()
{
       size--;
}

internal void Increase()
{
       size++;
}
```

La dernière méthode des tas est celle indiquant quelle est la probabilité de prendre un élément dans un tas. Elle suit le calcul expliqué lors de la déclaration de la constante.

```
    internal double ProbaToTake()
    {
        double proba = 1.0;
        for (int i = 1; i < size; i++)
        {
            proba *= PROBA_DECREASE;
        }
        return proba;
    }
}
```

Les tas de déchets sont complètement codés.

7.2.2 Les agents nettoyeurs

Les robots ou agents nettoyeurs héritent eux aussi de la classe `ObjectInWorld`. Cette classe **`WasteAgent`** possède de nombreux attributs supplémentaires :

- La charge actuellement portée, de type `Waste`.
- Le vecteur vitesse exprimé par ses coordonnées `speedX` et `speedY`.
- Une référence vers l'environnement dans lequel il évolue, qui est de type `WasteEnvironment` défini juste après.
- Un booléen indiquant s'il est actuellement occupé à poser ou prendre une charge, ou non.

On rajoute deux constantes : une indiquant la taille d'un pas (`STEP`) et l'autre la probabilité de changer de direction à chaque pas de temps (`CHANGE_DIRECTION_PROB`).

```
using System;
using System.Collections.Generic;
using System.Linq;

public class WasteAgent : ObjectInWorld
{
    protected const double STEP = 3;
    protected const double CHANGE_DIRECTION_PROB = 0.05;
```

```
    protected Waste load = null;
    protected double speedX;
    protected double speedY;
    protected WasteEnvironment env;
    protected bool busy = false;

    // Suite du code ici
}
```

Cette classe possède, comme pour les poissons de la simulation précédente, une méthode `Normalize` permettant de normaliser les vecteurs vitesse.

```
    private void Normalize()
    {
            double length = Math.Sqrt(speedX * speedX + speedY *
speedY);
            speedX = speedX / length;
            speedY = speedY / length;
    }
```

Le constructeur de la classe prend en paramètre la position en X et Y, ainsi que l'environnement qui a créé l'agent. La vitesse est choisie aléatoirement et normalisée.

```
    public WasteAgent(double _posX, double _posY,
WasteEnvironment _env)
    {
            PosX = _posX;
            PosY = _posY;
            env = _env;
            speedX = env.randomGenerator.NextDouble() - 0.5;
            speedY = env.randomGenerator.NextDouble() - 0.5;
            Normalize();
    }
```

Cette classe possède aussi une méthode indiquant si l'agent est chargé ou non.

```
    public bool isLoaded()
    {
            return load != null;
    }
```

La mise à jour de la position se fait par `UpdatePosition` qui prend en paramètres les coordonnées maximales de l'espace. Les positions sont incrémentées de la vitesse multipliée par le pas, et on vérifie ensuite que celles-ci ne sortent pas de la zone autorisée.

```
    public void UpdatePosition(double _maxWidth, double
_maxHeight) {
            PosX += STEP * speedX;
            PosY += STEP * speedY;
            if (PosX < 0)
            {
                  PosX = 0;
            }
            if (PosY < 0)
            {
                  PosY = 0;
            }
            if (PosX > _maxWidth)
            {
                  PosX = _maxWidth;
            }
            if (PosY > _maxHeight)
            {
                  PosY = _maxHeight;
            }
    }
```

La méthode la plus complexe est celle codant le comportement de l'agent, avec la modification de sa direction.

Tout d'abord, on cherche quelle est la zone de déchet ciblée par notre agent. Pour cela, on cherche si on est dans la zone d'un tas, et si on porte un déchet du même type que celui-ci.

Deux cas se présentent alors :

– Soit l'agent n'a pas de but potentiel (il est loin des tas) ou est occupé : dans ce cas-là, on choisit une nouvelle direction aléatoirement avec la probabilité `CHANGE_DIRECTION_PROB` définie précédemment, et si on est hors de toute zone, on note que l'on n'est plus occupé.

– Soit l'agent a un but potentiel : dans ce cas-là, on adapte notre direction vers le centre de la zone, et si on est proche du centre, alors soit on dépose la charge que l'on a (si on en a une), soit on prend un élément du tas avec une probabilité définie dans la classe `Waste`. Dans les deux cas, on prévient l'environnement via les méthodes `TakeWaste` et `SetWaste`, codées ultérieurement. De plus, on note que l'on est occupé.

Le booléen `busy` permet ainsi de ne pas redéposer de suite l'objet récupéré ou de reprendre un élément que l'on viendrait juste de poser, en s'assurant que l'on sorte de toute zone d'action avant de pouvoir agir de nouveau.

Quoi qu'il en soit, la nouvelle direction est normalisée pour garder une vitesse constante.

```
    internal void UpdateDirection(List<Waste> _wasteList)
    {
           // Où aller ?
           List<Waste> inZone = _wasteList.Where(x =>
DistanceTo(x) < x.Zone).OrderBy(x => DistanceTo(x)).ToList();
           Waste goal;
           if (load == null)
           {
                 goal = inZone.FirstOrDefault();
           }
           else
           {
                 goal = inZone.Where(x => x.Type ==
load.Type).FirstOrDefault();
           }

           // Avons-nous un but ?
           if (goal == null || busy)
           {
                 // Pas de but, se déplacer aléatoirement
                 if (env.randomGenerator.NextDouble() <
CHANGE_DIRECTION_PROB)
                 {
                         speedX = env.randomGenerator.NextDouble() - 0.5;
                         speedY = env.randomGenerator.NextDouble() - 0.5;
                 }
                 if (busy && goal == null)
                 {
                         busy = false;
                 }
           }
           else
```

```
            {
                // Aller au but
                speedX = goal.PosX - PosX;
                speedY = goal.PosY - PosY;

                // But atteint ? Prendre ou déposer une charge
                if (DistanceTo(goal) < STEP)
                {
                    if (load == null)
                    {
                        if
(env.randomGenerator.NextDouble() < goal.ProbaToTake())
                        {
                            load = env.TakeWaste(goal);
                        }
                    }
                    else
                    {
                        env.SetWaste(goal);
                        load = null;
                    }
                    busy = true;
                }
            }

            Normalize();
    }
```

Le comportement des agents se situe donc entièrement dans cette méthode. Il n'y a ni communication entre eux, ni aucune notion de plan ou de but à atteindre.

7.2.3 L'environnement

La dernière classe générique est **`WasteEnvironment`**, représentant notre environnement. Pour pouvoir prévenir l'interface qu'une mise à jour est prête, nous utilisons là aussi un `delegate`, et un évènement indiquant qu'elle est terminée.

Remarque

Pour plus d'explications sur ce délégué, se reporter à celui utilisé dans la classe Ocean de la simulation de bancs de poissons.

En plus du `delegate` défini avant la classe, et de l'évènement dans la classe, celle-ci possède plusieurs attributs :

- La liste des tas de déchets présents dans l'environnement.
- La liste des agents nettoyeurs.
- La taille de l'espace grâce à `MAX_WIDTH` et `MAX_HEIGHT`.
- Un générateur aléatoire.
- Le nombre d'itérations depuis le début.

Le code de base de la classe est donc le suivant :

```
using System;
using System.Collections.Generic;

public delegate void EnvironmentUpdated(List<Waste> _waste,
List<WasteAgent> _agents);

public class WasteEnvironment
{
    List<Waste> wasteList;
    List<WasteAgent> agents;
    public Random randomGenerator;
    double MAX_WIDTH;
    double MAX_HEIGHT;
    protected int nbIterations = 0;

    public event EnvironmentUpdated environmentUpdatedEvent;

    // Suite du code ici
}
```

Le constructeur prend en paramètre le nombre de déchets, d'agents, la taille de l'environnement et le nombre de types de déchets. Les tas comme les agents sont initialisés aléatoirement grâce au générateur aléatoire créé.

```
   public WasteEnvironment(int _nbWaste, int _nbAgents, double
_width, double _height, int _nbWasteTypes)
   {
           if (randomGenerator == null)
           {
                  randomGenerator = new Random();
           }
```

```
            MAX_WIDTH = _width;
            MAX_HEIGHT = _height;

            wasteList = new List<Waste>();
            for (int i = 0; i < _nbWaste; i++)
            {
                    Waste waste = new
Waste(randomGenerator.NextDouble() * MAX_WIDTH,
randomGenerator.NextDouble() * MAX_HEIGHT,
randomGenerator.Next(_nbWasteTypes));
                    wasteList.Add(waste);
            }

            agents = new List<WasteAgent>();
            for (int i = 0; i < _nbAgents; i++)
            {
                    WasteAgent agent = new
WasteAgent(randomGenerator.NextDouble() * MAX_WIDTH,
randomGenerator.NextDouble() * MAX_HEIGHT, this);
                    agents.Add(agent);
            }
    }
```

Nous avions vu que les agents avaient besoin de deux méthodes. La première, `SetWaste`, permet d'indiquer qu'un agent a déposé un nouvel élément sur un tas existant. Il suffit alors d'incrémenter la taille du tas visé :

```
    internal void SetWaste(Waste _goal)
    {
          _goal.Increase();
    }
```

La méthode `TakeWaste` vérifie la taille du tas : s'il n'y a qu'un élément, c'est celui-ci que l'agent va récupérer, et on supprime alors le tas de la liste des déchets. Si par contre le tas contient plusieurs éléments, on va juste le décrémenter et renvoyer un nouvel élément créé par copie (et donc ayant une charge de 1).

```
    internal Waste TakeWaste(Waste _goal)
    {
          if (_goal.Size == 1)
          {
                  wasteList.Remove(_goal);
                  return _goal;
```

```
            }
            else
            {
                    _goal.Decrease();
                    Waste load = new Waste(_goal);
                    return load;
            }
    }
```

La dernière méthode est celle mettant à jour l'environnement. Pour chaque agent, on lui demande de mettre à jour sa direction, puis sa position. Ensuite on incrémente le nombre d'itérations. Comme les agents sont "aspirés" par le premier tas à portée, il y a un biais. Toutes les 500 itérations on inverse donc l'ordre des tas, de manière à contrer ce biais. Enfin, une fois la mise à jour terminée, on déclenche l'évènement correspondant.

```
    public void Update()
    {
            foreach (WasteAgent agent in agents)
            {
                  agent.UpdateDirection(wasteList);
                  agent.UpdatePosition(MAX_WIDTH, MAX_HEIGHT);
            }

            nbIterations++;
            if (nbIterations % 500 == 0)
            {
                  wasteList.Reverse();
            }

            if (environmentUpdatedEvent != null)
            {
                  environmentUpdatedEvent(wasteList, agents);
            }
    }
```

7.2.4 L'application graphique

Il ne reste plus qu'à coder le programme principal, qui sera contenu dans le code-behind de la fenêtre **MainWindow**. Celle-ci, de 400 par 600 pixels, ne contient qu'un `canvas`. Son code XAML est donc le suivant :

```
<Window x:Class="MainWindow"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Waste sorting" Height="400" Width="600">
    <Canvas Name="environmentCanvas" Height="Auto" Width="Auto"
Background="White"/>
</Window>
```

Pour le code-behind, la classe **MainWindow** va garder un pointeur vers l'environnement créé pour pouvoir appeler les mises à jour et un `timer`. Celui-ci pourra être arrêté ou relancé à chaque clic dans la fenêtre, ce qui permettra de mettre le programme en pause. On conserve donc aussi un booléen `play` indiquant si le `timer` est actuellement en route.

```
using MultiAgentSystemPCL;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Windows;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Shapes;
using System.Windows.Threading;

public partial class MainWindow : Window
{
    WasteEnvironment environment;
    bool play = false;
    DispatcherTimer updateTimer;

    // Suite du code ici
}
```

Le constructeur, en plus d'initialiser la fenêtre, s'abonne aux évènements Loaded de la fenêtre et au clic de la souris dans le canvas. Enfin, elle crée un nouveau timer, qui est programmé pour se lancer toutes les 10ms (mais qui est arrêté au départ).

```
    public MainWindow()
    {
            InitializeComponent();
            Loaded += MainWindow_Loaded;
            environmentCanvas.MouseDown +=
environmentCanvas_MouseDown;

            updateTimer = new DispatcherTimer();
            updateTimer.Tick += dispatcherTimer_Tick;
            updateTimer.Interval = new TimeSpan(0, 0, 0, 0, 10);
    }
```

Lorsque la fenêtre est chargée, on crée un environnement (avec 50 déchets de 3 types pour 30 agents) et on lance manuellement la première mise à jour pour avoir un affichage de l'état initial.

```
    void MainWindow_Loaded(object _sender, RoutedEventArgs _e)
    {
            environment = new WasteEnvironment(50, 30,
environmentCanvas.ActualWidth, environmentCanvas.ActualHeight, 3);
            environment.environmentUpdatedEvent +=
environment_environmentUpdatedEvent;
            environment.Update();
    }
```

Lorsque l'on clique avec la souris, on lance ou arrête le timer selon son état actuel :

```
    void environmentCanvas_MouseDown(object _sender,
MouseButtonEventArgs _mouseEvent)
    {
            if (play)
            {
                    updateTimer.Stop();
            }
            else
            {
                    updateTimer.Start();
            }
            play = !play;
    }
```

De plus, lorsque le `timer` se déclenche, on va appeler la mise à jour de l'environnement :

```
void dispatcherTimer_Tick(object _sender, EventArgs _e)
{
        environment.Update();
}
```

Il faut ensuite plusieurs méthodes graphiques. Tout d'abord, les agents sont dessinés sous la forme d'un carré de 3 px de côté. Celui-ci est noir s'il ne transporte rien et gris s'il transporte un élément.

```
private void DrawAgent(WasteAgent _agent)
{
        Rectangle rect = new Rectangle();
        rect.Width = 3;
        rect.Height = 3;
        rect.Margin = new Thickness(_agent.PosX - 1,
_agent.PosY - 1, 0, 0);
        if (_agent.isLoaded())
        {
                rect.Stroke = Brushes.Gray;
                rect.Fill = Brushes.Gray;
        }
        else
        {
                rect.Stroke = Brushes.Black;
                rect.Fill = Brushes.Black;
        }
        environmentCanvas.Children.Add(rect);
}
```

Pour les tas, ils sont représentés sous la forme d'un carré de 3 px de côté, entouré d'une zone indiquant la portée de celui-ci. Cela permettra de voir où sont les zones d'influence et aussi de visualiser la taille d'un tas de manière graphique. De plus, nous utilisons trois couleurs différentes pour les trois types de déchets.

```
private void DrawWaste(Waste _waste)
{
        Rectangle rect = new Rectangle();
        rect.Width = 3;
        rect.Height = 3;
        rect.Margin = new Thickness(_waste.PosX - 1,
```

```
_waste.PosY - 1, 0, 0);
            Brush strokeAndFill = Brushes.Transparent;
            switch (_waste.Type)
            {
                  case 0:
                          strokeAndFill = Brushes.Red;
                          break;
                  case 1:
                          strokeAndFill = Brushes.Blue;
                          break;
                  case 2:
                          strokeAndFill = Brushes.ForestGreen;
                          break;
            }
            rect.Stroke = strokeAndFill;
            rect.Fill = strokeAndFill;
            environmentCanvas.Children.Add(rect);

            Ellipse zone = new Ellipse();
            zone.Width = 2 * _waste.Zone;
            zone.Height = 2 * _waste.Zone;
            zone.Margin = new Thickness(_waste.PosX - _waste.Zone,
_waste.PosY - _waste.Zone, 0, 0);
            zone.Fill = strokeAndFill;
            zone.Opacity = 0.3;
            environmentCanvas.Children.Add(zone);
    }
```

La dernière méthode est celle qui est déclenchée par l'évènement de fin de mise à jour de l'environnement. On affiche en console le nombre de tas et le nombre d'agents. On efface ensuite le canvas, puis on dessine chaque tas et chaque agent. S'il ne reste plus que trois tas, et que tous les agents ont posé leur charge, alors on a réussi le tri et on arrête la simulation.

```
    void environment_environmentUpdatedEvent(List<Waste> _wasteList,
List<WasteAgent> _agentList)
    {
            int nbWaste = _wasteList.Count();
            int nbAgentsLoaded = _agentList.Where(x =>
x.isLoaded()).Count();

            environmentCanvas.Children.Clear();

            foreach (Waste waste in _wasteList)
```

```
            {
                DrawWaste(waste);
            }

            foreach (WasteAgent agent in _agentList)
            {
                DrawAgent(agent);
            }

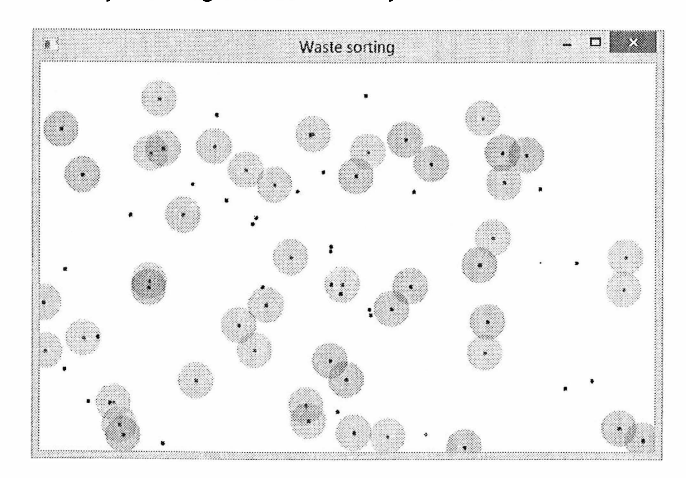
            Console.Out.WriteLine(nbWaste + " - " + nbAgentsLoaded);
            if (nbWaste == 3 && nbAgentsLoaded == 0)
            {
                updateTimer.Stop();
            }
        }
```

Rien ne garantit que l'on converge effectivement vers trois tas (un par type) contenant tous les déchets de ce type ; pourtant en pratique c'est ce qui se produit.

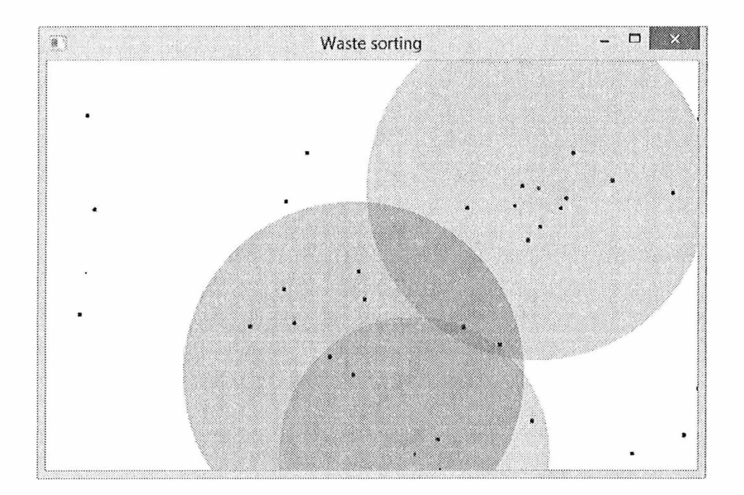
7.2.5 Résultats obtenus

À part quelques simulations pendant lesquelles les agents enlèvent tous les déchets d'un type, ce qui ne permet plus de les reposer ultérieurement, toutes les autres simulations convergent vers la présence de trois tas, un par type.

Dans une simulation typique, on a le départ suivant (les déchets sont entourés alors que les agents sont des petits carrés seuls) :



Après plusieurs secondes à quelques minutes de simulation, les déchets ont été déplacés par les agents et nous nous retrouvons dans la configuration suivante :



On voit alors qu'il n'y a plus que trois tas, un par type. On peut aussi voir que sur cette simulation, il existe un tas plus petit que les deux autres, tout simplement car les déchets sont répartis aléatoirement entre les différents types.

Le comportement de tri a cependant bien été obtenu, uniquement par émergence.

7.3 Le jeu de la vie

Cette application est un petit jeu de la vie. Des cellules, initialisées aléatoirement pour commencer, vont évoluer dans une grille en suivant les règles de Conway.

On observe une stabilisation au bout de plusieurs générations, avec des structures qui n'évoluent plus et d'autres qui évoluent selon un cycle entre plusieurs positions possibles.

L'utilisateur peut à tout moment mettre l'application en pause ou la relancer, et rajouter ou enlever des cellules vivantes en cliquant sur la fenêtre.

7.3.1 La grille

Les agents sont ici très simples, car il s'agit simplement de cellules ne pouvant pas se déplacer et ne prenant que deux états possibles : vivante ou morte. Nous n'allons donc pas les coder dans une classe à part, mais directement dans la grille, qui représente l'environnement. De plus, il n'existe aucun autre objet.

La grille **GoLGrid** (GoL pour *Game of Life*) est dessinée à chaque mise à jour, aussi nous déclarons là encore un `delegate`, et un évènement dans la classe pour indiquer que celle-ci est terminée.

La grille possède en plus une largeur et une hauteur en nombre de cellules et un tableau à deux dimensions contenant toutes les cellules, qui sont simplement des booléens indiquant si la cellule située sur cette case est en vie.

La base du code est donc la suivante :

```
using System;

public delegate void gridUpdated(bool[][] newGrid);
public class GoLGrid
{
    protected int WIDTH;
    protected int HEIGHT;
    bool[][] gridCells = null;

    public event gridUpdated gridUpdatedEvent = null;

    // Suite du code ici
}
```

Le constructeur prend en paramètres la largeur et la hauteur de la grille ainsi que la densité en cellules vivantes au départ. Celles-ci sont initialisées aléatoirement grâce à un générateur aléatoire en même temps que le tableau de cellules est créé.

```
    public GoLGrid(int _width, int _height, double _cellDensity)
    {
        WIDTH = _width;
        HEIGHT = _height;

        Random randomGen = new Random();
```

```
        gridCells = new bool[WIDTH][];
        for(int i = 0; i < WIDTH; i++) {
               gridCells[i] = new bool[HEIGHT];
               for (int j = 0; j < HEIGHT; j++)
               {
                         if (randomGen.NextDouble() <
_cellDensity)
                         {
                                gridCells[i][j] = true;
                         }
               }
        }
  }
```

Nous avons besoin de deux méthodes pour pouvoir faire la mise à jour de la grille : une permettant de changer l'état d'une cellule donnée, et une permettant de connaître le nombre de voisines vivantes.

Pour changer l'état d'une cellule, il suffit d'inverser la valeur du booléen :

```
  public void ChangeState(int _row, int _col)
  {
         gridCells[_row][_col] = !gridCells[_row][_col];
  }
```

Pour compter le nombre de cellules voisines vivantes, il faut regarder dans la zone adjacente à la cellule (donc une zone de 3*3 centrée sur la cellule). Cependant, il faut faire attention à ne pas sortir de la grille, aussi on commence par vérifier les coordonnées minimales et maximales testées par rapport aux dimensions de l'environnement. Il faut aussi penser à ne pas compter la cellule au centre.

```
  private int NbCellAround(int _cellRow, int _cellCol)
  {
         int count = 0;

         int row_min = (_cellRow - 1 < 0 ? 0 : _cellRow - 1);
         int row_max = (_cellRow + 1 > WIDTH-1 ? WIDTH-1 :
_cellRow + 1);
         int col_min = (_cellCol - 1 < 0 ? 0 : _cellCol - 1); ;
         int col_max = (_cellCol + 1 > HEIGHT-1 ? HEIGHT-1 :
_cellCol + 1); ;

         for (int row = row_min; row <= row_max; row++)
```

```
            {
                  for (int col = col_min; col <= col_max; col++)
                  {
                        if (gridCells[row][col] && !(row ==
_cellRow && col == _cellCol)) {
                              count++;
                        }
                  }
            }
            return count;
      }
```

La dernière méthode est la mise à jour. Pour cela, on commence par créer une nouvelle grille vierge (avec toutes les cellules considérées comme mortes). On parcourt ensuite la grille, et pour chaque cellule, on compte son nombre de voisines dans la grille actuelle :

- Si elle a trois voisines, elle sera vivante dans la prochaine grille.
- Si elle a deux voisines et qu'elle était vivante, alors elle le restera.
- Dans tous les autres cas, elle sera morte (et donc on n'a rien à faire).

On remplace ensuite l'ancienne grille par la nouvelle calculée et on déclenche l'évènement indiquant la fin de la mise à jour.

On rajoute à cette méthode un booléen en paramètre indiquant si on souhaite une mise à jour réelle de l'application (par défaut) ou simplement si on souhaite lancer l'évènement pour récupérer l'état actuel des cellules. Ce dernier cas sera utilisé lorsque l'utilisateur voudra changer l'état d'une cellule pour mettre à jour l'affichage.

```
      public void Update(bool _withUpdate = true)
      {
            if (_withUpdate)
            {
                  bool[][] newGrid = new bool[WIDTH][];
                  for (int i = 0; i < WIDTH; i++)
                  {
                        newGrid[i] = new bool[HEIGHT];
                        for (int j = 0; j < HEIGHT; j++)
                        {
                              int count = NbCellAround(i, j);
                              if (count == 3 || (count == 2 &&
```

```
gridCells[i][j]))
                              {
                                    newGrid[i][j] = true;
                              }
                        }
                  }

                  gridCells = newGrid;
            }

            if (gridUpdatedEvent != null)
            {
                  gridUpdatedEvent(gridCells);
            }
      }
```

7.3.2 L'application graphique

Comme pour les deux autres programmes, la fenêtre principale **MainWindow** est très simple, ne contenant qu'un `canvas` de 300 * 600 px.

```
<Window x:Class="MainWindow"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Conway's Game of Life" Height="339" Width="616">
    <Canvas Name="gridCanvas" Height="300" Width="600"
Background="White"/>
</Window>
```

Le code-behind contient là encore toute la logique du code, pour des raisons de lisibilité. La classe **MainWindow** contient un `timer` pour lancer les mises à jour, un booléen indiquant si celles-ci sont lancées ou en pause (car on pourra arrêter ou relancer la simulation d'un clic gauche) et une référence vers la grille.

```
using MultiAgentSystemPCL;
using System;
using System.Linq;
using System.Windows;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Shapes;
using System.Windows.Threading;
```

```
public partial class MainWindow : Window
{
    DispatcherTimer updateTimer;
    bool running = false;
    GoLGrid grid;

    // Suite du code ici
}
```

Le constructeur initialise les composants graphiques, puis s'abonne à deux évènements : `loaded` qui indique que le chargement de la fenêtre est terminé, et `MouseDown` qui indique le clic de la souris (droit ou gauche).

```
    public MainWindow()
    {
          InitializeComponent();
          Loaded += MainWindow_Loaded;
          gridCanvas.MouseDown += gridCanvas_MouseDown;
    }
```

Lorsque la fenêtre est chargée, on crée une nouvelle grille, et on s'abonne à son évènement de mise à jour. On crée aussi un `timer` qui se lancera toutes les 500ms, et on lance la simulation.

```
    void MainWindow_Loaded(object _sender, RoutedEventArgs _e)
    {
          grid = new GoLGrid((int) gridCanvas.ActualWidth / 3,
(int) gridCanvas.ActualHeight / 3, 0.1);
          grid.gridUpdatedEvent += grid_gridUpdatedEvent;

          updateTimer = new DispatcherTimer();
          updateTimer.Tick += updateTimer_Tick;
          updateTimer.Interval = new TimeSpan(0, 0, 0, 0, 500);

          updateTimer.Start();
          running = true;
    }
```

À chaque fois que le `timer` se déclenche, on lance la mise à jour de la grille :

```
    void updateTimer_Tick(object _sender, EventArgs _e)
    {
          grid.Update();
    }
```

Lorsqu'un clic de souris se produit, on regarde tout d'abord quel est le bouton utilisé :

- S'il s'agit du bouton gauche, alors on change l'état de la cellule située sous le clic (en prenant en compte le fait que chaque cellule est représentée par 3 px en largeur et 3 px en hauteur). On demande alors une mise à jour de l'affichage, sans recalculer les cellules vivantes ou mortes.
- S'il s'agit du bouton droit, alors on lance ou stoppe la simulation selon son état actuel.

```
   void gridCanvas_MouseDown(object _sender,
MouseButtonEventArgs _mouseEvent)
   {
         if (_mouseEvent.LeftButton == MouseButtonState.Pressed)
         {

   grid.ChangeState((int)(_mouseEvent.GetPosition(gridCanvas).X
/ 3), (int)(_mouseEvent.GetPosition(gridCanvas).Y / 3));
                 grid.Update(false);
         }
         else if (_mouseEvent.RightButton == MouseButtonState.Pressed)
         {
                 if (running)
                 {
                        updateTimer.Stop();
                 }
                 else
                 {
                        updateTimer.Start();
                 }
                 running = !running;
         }

   }
```

Pour les méthodes graphiques, le dessin d'une cellule vivante consiste juste à dessiner un carré de 3 px de côté en noir.

```
private void DrawCell(int _row, int _col)
{
      Rectangle rect = new Rectangle();
      rect.Width = 3;
      rect.Height = 3;
      rect.Margin = new Thickness(3 * _row, 3 * _col, 0, 0);
      rect.Stroke = Brushes.Black;
      rect.Fill = Brushes.Black;

      gridCanvas.Children.Add(rect);
}
```

La dernière méthode est celle se déclenchant suite à la mise à jour. On efface l'écran actuel, puis on parcourt la grille, et on dessine les cellules vivantes.

```
void grid_gridUpdatedEvent(bool[][] _grid)
{
      gridCanvas.Children.Clear();

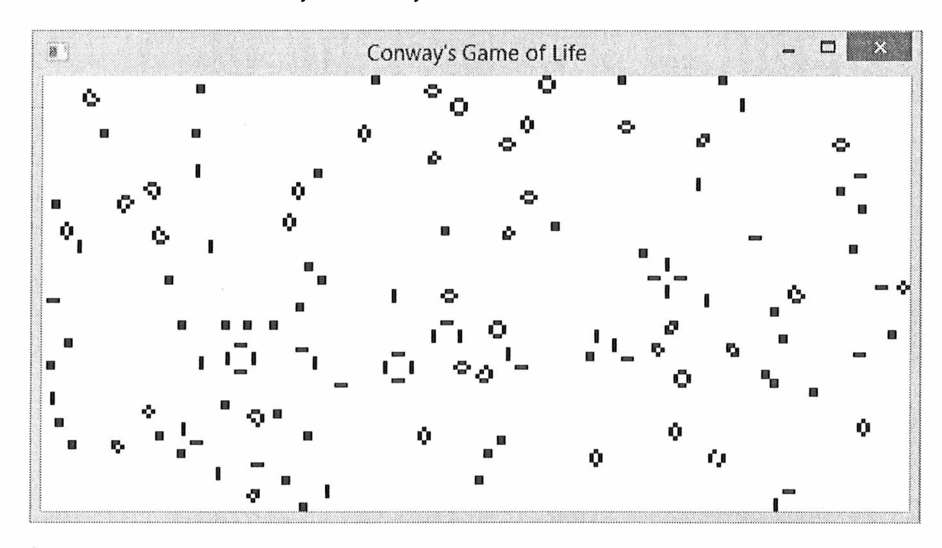
      for(int row = 0; row < _grid.Count(); row++) {
            for (int col = 0; col < _grid[0].Count(); col++)
            {
                  if (_grid[row][col])
                  {
                        DrawCell(row, col);
                  }
            }
      }
}
```

Le jeu de la vie est alors opérationnel. Il est très rapide à coder, vu qu'il tient dans deux classes assez simples.

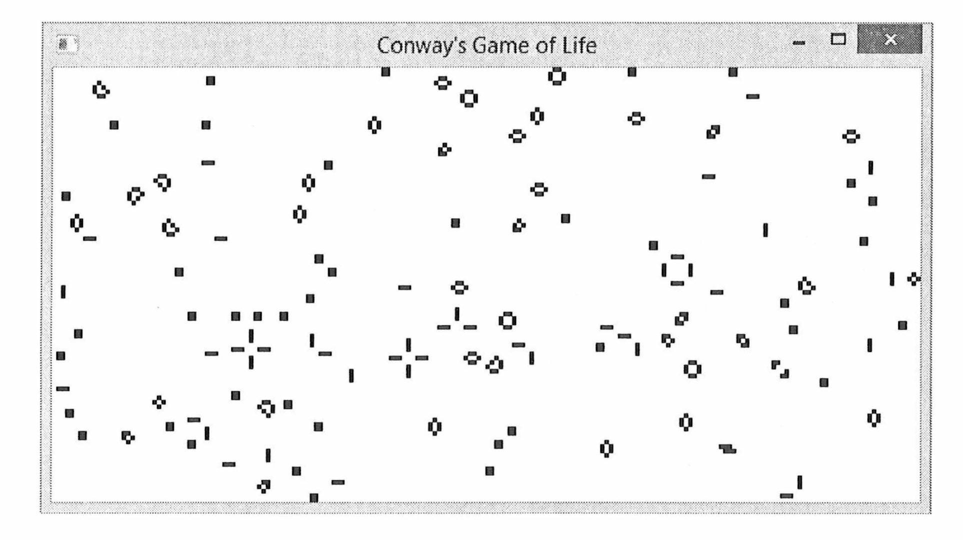
7.3.3 Résultats obtenus

Lors de la deuxième itération, toutes les cellules isolées disparaissent. Au cours des itérations suivantes, des "explosions" ont lieu sur la grille, laissant derrière elles des structures stables ou oscillantes, majoritairement de période 2. De temps en temps apparaissent aussi quelques vaisseaux qui se déplacent jusqu'à rencontrer une autre structure.

Au bout de plusieurs générations, seules les structures stables et oscillantes restent visibles. Voici par exemple l'état final d'une simulation à l'itération N :



À l'itération N+1, on obtient la fenêtre suivante :



L'itération suivante (N+2) est la même que l'itération N : notre grille ne contient que des structures stables (dont les blocs, mares et ruches présentées avant) et des structures oscillantes de période 2 : des clignotants (souvent situés en croix ou en cercle selon le temps), des crapauds, et plus rarement des bateaux.

Dans de rares cas, quelques autres structures peuvent apparaître.

L'utilisateur peut à tout moment rajouter ou enlever des cellules vivantes. La grille va alors évoluer jusqu'à se stabiliser de nouveau.

8. Synthèse

Les systèmes multi-agents permettent de résoudre un grand nombre de problèmes, à la fois dans la simulation de foules, dans la planification et la recherche de chemins et dans la simulation de problèmes complexes, pour mieux les comprendre et les étudier.

Ils reposent tous sur les observations faites sur les insectes eusociaux, qui sont capables de résoudre des tâches très complexes à partir de règles très simples. C'est par émergence que la solution apparaît, et non par un plan préprogrammé. Les abeilles trouvent de nouvelles sources de nourriture, les fourmis communiquent par phéromones pour optimiser les accès à la nourriture et les termites construisent d'énormes termitières climatisées.

En informatique, les systèmes multi-agents contiennent donc un environnement dans lequel se trouvent des objets et des agents. Il n'y a que très peu de règles à suivre sur ceux-ci, et chaque problème pourra donc avoir une ou plusieurs modélisations possibles.

Il existe cependant quelques algorithmes plus connus parmi les systèmes multi-agents. On peut citer les algorithmes simulant les comportements de meutes basés sur les boids, l'optimisation par colonies de fourmis et ses phéromones artificielles, les systèmes immunitaires artificiels permettant de détecter et de réagir à des attaques ou menaces et les automates à états finis, dont le plus connu est le jeu de la vie de Conway.

Dans tous les cas, c'est la multiplication des agents et les liens qu'ils ont, directement par communication ou indirectement par stigmergie (ou même sans communication entre eux) qui va faire émerger la solution. On observe alors la puissance de l'intelligence distribuée.

Dans un monde où il existe de plus en plus d'éléments connectés via Internet, on comprend vite que ces systèmes multi-agents ont un grand avenir et de nombreuses possibilités non encore exploitées. On pourrait ainsi faire communiquer des objets connectés et en sortir des comportements intelligents nous aidant au quotidien.

Chapitre 7
Réseaux de neurones

1. Présentation du chapitre

L'intelligence artificielle a longtemps eu pour but de simuler l'intelligence humaine, et d'obtenir un système artificiel capable de réflexion, de prise de décision et d'apprentissage.

Les chercheurs se sont donc assez rapidement intéressés au fonctionnement du cerveau pour le reproduire. C'est ainsi que les premiers neurones artificiels ont été définis par Mac Culloch et Pitts en 1943.

Aujourd'hui, on ne cherche plus à créer des cerveaux avec toutes leurs capacités, mais à avoir des systèmes pouvant résoudre certains problèmes complexes sur lesquels les systèmes classiques sont limités. C'est ainsi que sont nés les **réseaux de neurones artificiels**.

Ce chapitre commence par en expliquer les origines biologiques, en s'intéressant au fonctionnement des encéphales, et plus précisément aux neurones.

Le neurone formel est ensuite présenté. Le perceptron, un des modèles les plus simples de réseaux, ainsi que son apprentissage sont expliqués.

Cependant, il est insuffisant pour résoudre de nombreux problèmes. Les réseaux de type "feed-forward", plus puissants, sont alors présentés, avec en particulier les réseaux dits MLP (*MultiLayer Perceptron*) et RBF (*Radial Basis Function*).

Le chapitre continue avec une présentation des différentes formes d'apprentissage de ces systèmes. Enfin, d'autres types de réseaux et les principaux domaines d'application sont exposés pour terminer cette partie théorique.

Une implémentation d'un réseau MLP avec apprentissage est proposée en C#, ainsi que son application à deux problèmes différents. Une synthèse clôt ce chapitre.

2. Origine biologique

On sait depuis longtemps que la réflexion se fait grâce au **cerveau**. Celui-ci a donc été étudié assez tôt (dès le 18e siècle).

Il existe des "cartes" du cerveau, indiquant ses principales structures et leurs rôles associés. Si tout n'est pas encore compris, on sait par exemple que le cervelet est très important pour la coordination des mouvements ou que l'hypothalamus gère des fonctions importantes comme le sommeil, la faim ou la soif.

Remarque

Contrairement à une idée reçue, que l'on retrouve même dans des films récents comme Lucy de Luc Besson sorti en 2014, on utilise bien 100 % de notre cerveau. Cependant, à un moment donné, seule une partie de celui-ci est mobilisée, en fonction des besoins. Une zone qui ne serait pas utilisée subirait une forte dégénérescence et disparaîtrait rapidement.

Les cellules les plus importantes du cortex cérébral sont les **neurones**. Ceux-ci sont très nombreux, vu que l'on en compte presque une centaine de milliards chez l'être humain. Ces cellules demandant énormément d'énergie et étant fragiles, elles sont protégées et nourries par les cellules gliales (90 % des cellules du cerveau), qui n'ont par contre aucun rôle dans la réflexion.

On sait que les neurones communiquent entre eux via des impulsions électriques. En effet, les "capteurs" (œil, oreille, peau...) envoient des impulsions électriques aux neurones, qu'ils traitent et transmettent ou non aux autres cellules.

Chaque neurone possède donc autour de son cœur (nommé soma) :

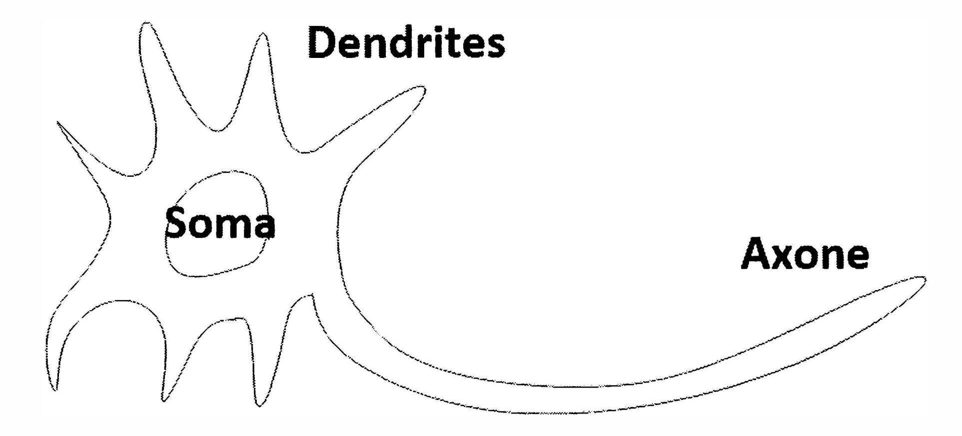
- Des **dendrites**, qui sont ses entrées.
- Un long **axone** lui servant de sortie.

Les signaux électriques arrivent donc au soma en suivant les dendrites, puis sont traités : selon l'intensité et la somme des impulsions reçues, le neurone envoie ou non une impulsion le long de son axone. Celui-ci est relié à des dendrites d'autres neurones.

Remarque

Le lien physique entre deux neurones se fait grâce aux synapses, cependant leur fonctionnement n'est pas expliqué plus en détail, n'étant pas utile à la compréhension des réseaux artificiels.

Un neurone peut donc être schématisé comme suit (l'axone possède en fait des ramifications lui permettant de se connecter à plusieurs autres neurones) :



Chaque neurone est donc une entité très simple, faisant simplement un travail sur les impulsions reçues pour choisir ou non d'en envoyer une en sortie. La puissance du cerveau se situe en fait dans le nombre de neurones et les nombreuses interconnexions entre eux.

3. Le neurone formel

Le neurone artificiel, aussi appelé **neurone formel**, reprend le fonctionnement du neurone biologique.

3.1 Fonctionnement général

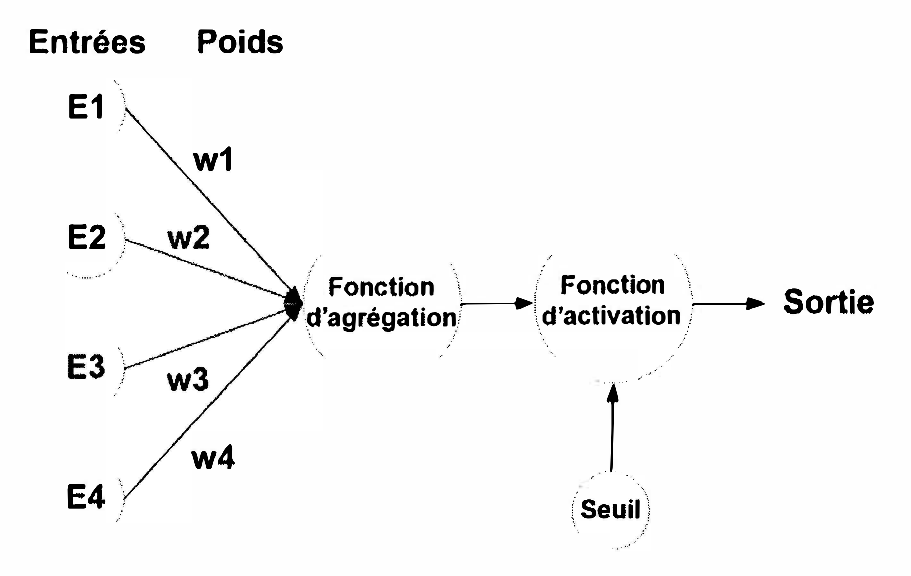
Un neurone reçoit des entrées et fournit une sortie, grâce à différentes caractéristiques :

- Des **poids** accordés à chacune des entrées, permettant de modifier l'importance de certaines par rapport aux autres.
- Une **fonction d'agrégation**, qui permet de calculer une unique valeur à partir des entrées et des poids correspondants.
- Un **seuil** (ou biais), permettant d'indiquer quand le neurone doit agir.
- Une **fonction d'activation**, qui associe à chaque valeur agrégée une unique valeur de sortie dépendant du seuil.

Remarque

La notion de temps, importante en biologie, n'est pas prise en compte pour la majorité des neurones formels.

Le neurone formel peut donc se résumer sous la forme suivante :



3.2 Fonctions d'agrégation

Il est possible d'imaginer plusieurs fonctions d'agrégation. Les deux plus courantes sont :

– La somme pondérée.

– Le calcul de distance.

Dans le cas de la **somme pondérée**, on va simplement faire la somme de toutes les entrées multipliées par leur poids. Mathématiquement cela s'exprime sous la forme :

$$\sum_{i=1}^{n} E_i * w_i$$

Dans le deuxième cas, celui du **calcul des distances**, on va comparer les entrées aux poids (qui sont les entrées attendues par le neurone), et calculer la distance entre les deux.

Pour rappel, la distance est la racine de la somme des différences au carré, ce qui s'exprime donc :

$$\sqrt{\sum_{i=1}^{n}(E_i - w_i)^2}$$

D'autres fonctions d'agrégation peuvent bien sûr être imaginées. L'important est d'associer une seule valeur à l'ensemble des entrées et des poids grâce à une fonction linéaire.

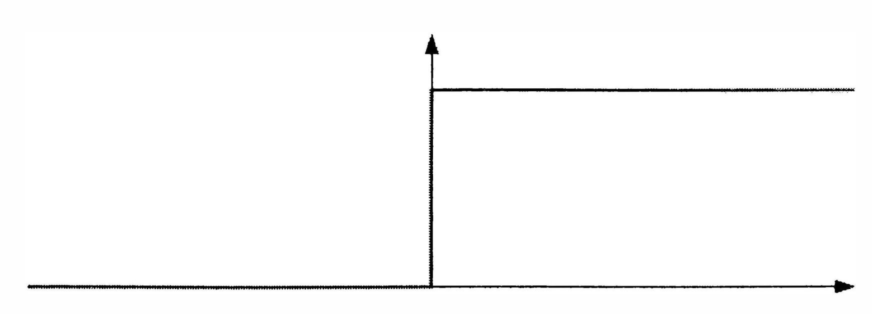
3.3 Fonctions d'activation

Une fois une valeur unique calculée, le neurone compare cette valeur à un seuil et en décide la sortie. Pour cela, plusieurs fonctions peuvent être utilisées. Les trois plus utilisées sont ici présentées.

3.3.1 Fonction "heavyside"

La fonction signe, ou **heavyside** en anglais, est une fonction très simple : elle renvoie +1 ou 0.

Ainsi, si la valeur agrégée calculée est plus grande que le seuil, elle renvoie +1, sinon 0 (ou -1 selon les applications).



Cette fonction permet par exemple la classification, en indiquant qu'un objet est ou non dans une classe donnée. Elle peut aussi être mise en place pour d'autres applications, mais elle reste parfois difficile à utiliser, car elle n'indique pas à quel point une valeur est forte. Elle peut donc ralentir l'apprentissage.

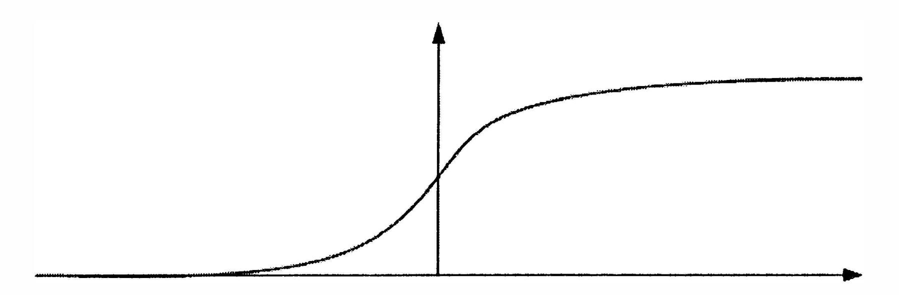
3.3.2 Fonction sigmoïde

La fonction **sigmoïde** utilise une exponentielle. Elle est définie par :

$$f(x) = \frac{1}{1 + e^{-x}}$$

Elle est comprise entre 0 et +1, avec une valeur de 0.5 en 0.

Dans le neurone, la méthode est appelée avec x = valeur agrégée - seuil. Ainsi, on a une sortie supérieure 0.5 si la valeur agrégée est plus grande que le seuil, inférieure à 0.5 sinon.



Cette fonction permet un meilleur apprentissage, grâce à sa pente. En effet, il est plus facile de savoir vers quelle direction aller pour améliorer les résultats, contrairement à la fonction heavyside qui n'est pas dérivable.

La dérivée de la sigmoïde, utilisée lors de l'apprentissage, est :

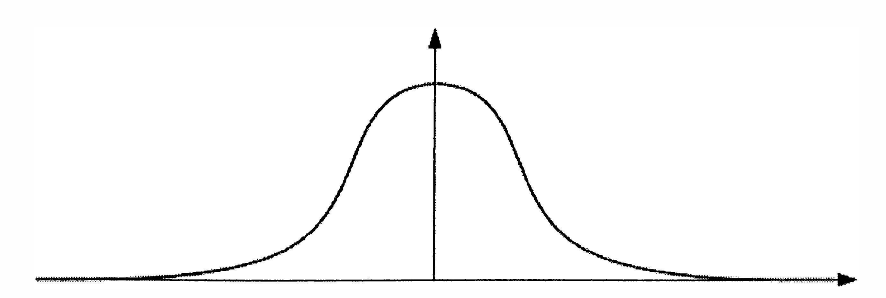
$$f'(x) = f(x) \,.\, (1 - f(x))$$

3.3.3 Fonction gaussienne

La dernière fonction très utilisée est la fonction **gaussienne**. Celle-ci, aussi appelée "courbe en cloche", est symétrique, avec un maximum obtenu en 0.

Son expression est plus complexe que pour la fonction sigmoïde, mais elle peut se simplifier sous la forme suivante, avec k et k' des constantes dépendant de l'écart-type voulu :

$$f(x) = k.e^{-x^2/k'}$$



Là encore, on utilisera la différence entre la valeur agrégée et le seuil comme abscisse.

Cette fonction étant aussi dérivable, elle permet un bon apprentissage. Cependant, contrairement aux fonctions précédentes, elle n'a qu'un effet local (autour du seuil) et non sur l'espace de recherche complet. Selon les problèmes à résoudre, cela peut être un avantage ou un inconvénient.

3.4 Poids et apprentissage

Les neurones formels sont tous identiques. Ce qui va les différencier, ce sont les seuils de chacun ainsi que les poids les liant à leurs entrées.

Sur des fonctions simples, il est possible de déterminer les poids et les seuils directement, cependant ce n'est jamais le cas lorsqu'un réseau de neurones est vraiment utile (donc sur des problèmes complexes).

L'**apprentissage** va donc consister à trouver pour chaque neurone du réseau les meilleures valeurs pour obtenir la sortie attendue. Plus un neurone a d'entrées donc plus il va y avoir de poids à ajuster, et plus l'apprentissage sera complexe et/ou long.

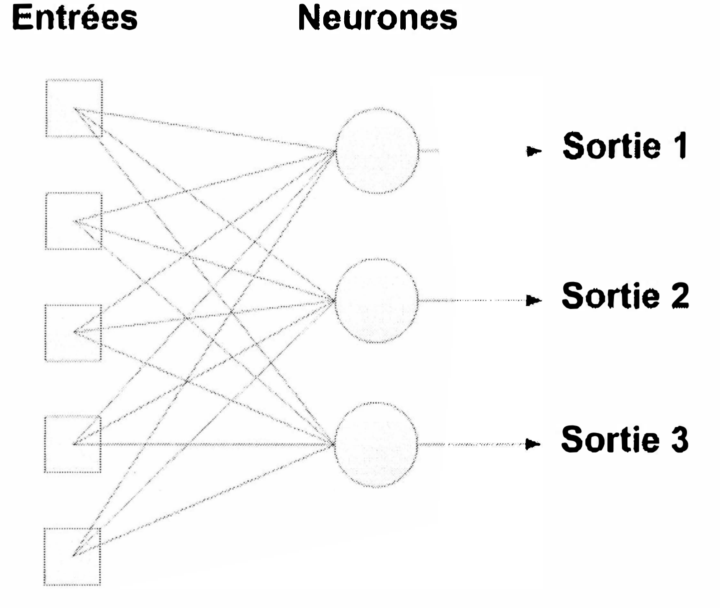
4. Perceptron

Le **perceptron** est le plus simple des réseaux de neurones.

4.1 Structure

Un perceptron est un réseau contenant p neurones. Chacun est relié aux n entrées. Ce réseau permet d'avoir p sorties. Généralement, chacune représente une décision ou une classe, et c'est la sortie ayant la plus forte valeur qui est prise en compte.

Avec 3 neurones et 5 entrées, on a donc 3 sorties. Voici la structure obtenue dans ce cas :



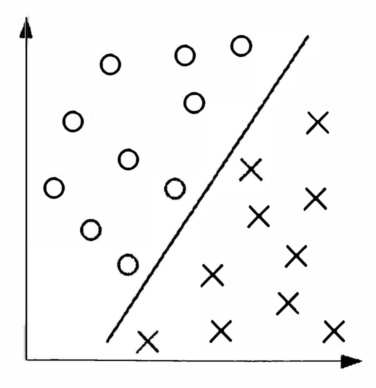
Le réseau possède alors 3 * 5 = 15 poids à ajuster, auxquels s'ajoutent 3 valeurs seuils (une par neurone).

4.2 Condition de linéarité

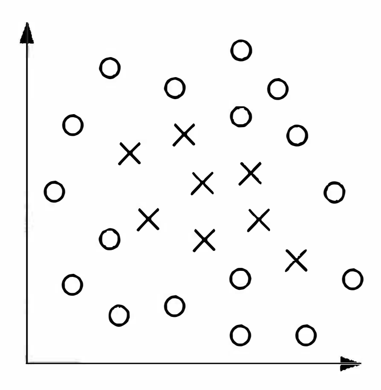
Le perceptron est simple, et donc facile à mettre en œuvre, mais il a une limite importante : seuls les **problèmes linéairement séparables** peuvent être résolus par celui-ci.

En effet, les fonctions d'activation utilisées (généralement heavyside, plus rarement sigmoïde) présentent un seuil, séparant deux zones de l'espace.

Imaginons un problème possédant deux classes, des croix et des ronds. Si les deux classes sont disposées comme suit, alors le problème est bien linéairement séparable. Une séparation possible est indiquée.



Au contraire, si les points sont présentés comme suit, les classes ne sont pas linéairement séparables et un réseau de type perceptron ne pourra pas résoudre ce problème.



Il faut donc, avant d'utiliser un réseau de type perceptron, s'assurer que le problème pourra être résolu. Sinon il faudra opter pour des réseaux plus complexes.

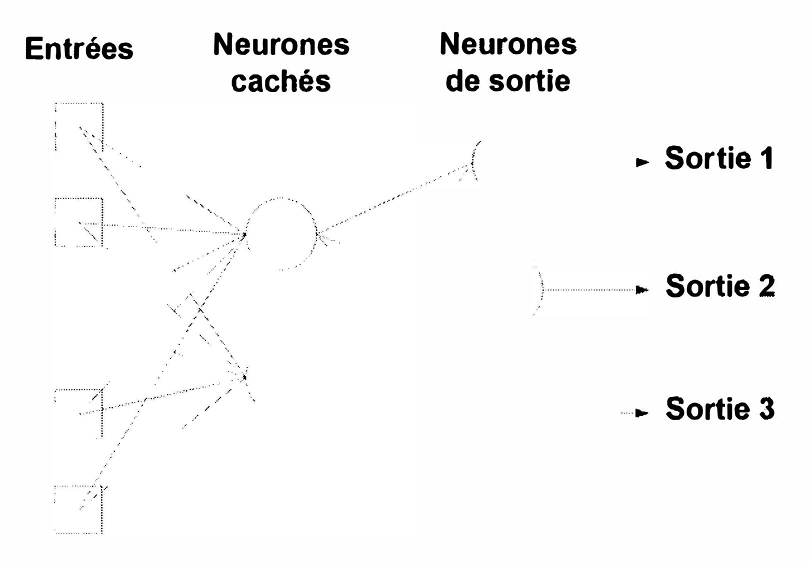
5. Réseaux feed-forward

Les réseaux de type "**feed-forward**" ou **à couches** permettent de dépasser les limites des perceptrons. En effet, ceux-ci ne sont plus limités aux problèmes linéairement séparables.

Ils sont composés d'une ou plusieurs couches cachées de neurones, reliées aux entrées ou aux couches précédentes, et une couche de sortie, reliée aux neurones cachés. On les appelle feed-forward car l'information ne peut aller que des entrées aux sorties, sans revenir en arrière.

Il est possible de trouver des réseaux avec plusieurs couches cachées, cependant ces réseaux apportent plus de complexité pour des capacités équivalentes à des réseaux à une seule couche cachée. Ce sont donc ces derniers qui sont les plus utilisés.

On obtient le réseau suivant si l'on a 5 entrées et 3 sorties, avec 2 neurones cachés.



Dans ce cas-là, il faut ajuster les poids et seuils de tous les neurones cachés (ici 12 paramètres) ainsi que les poids et seuils des neurones de sortie (9 paramètres). Le problème complet contient donc 21 valeurs à déterminer.

De plus, aucune règle ne permet de connaître le nombre de neurones cachés idéal pour un problème donné. Il est donc nécessaire de tester plusieurs valeurs et de choisir celle donnant les meilleurs résultats.

Les réseaux utilisant des neurones de type perceptron sont dits **MLP** pour *MultiLayer Perceptron*, alors que ceux utilisant des neurones à fonction d'activation gaussienne sont dit **RBF** (pour *Radial Basis Function*). Les réseaux MLP et RBF sont les plus courants.

6. Apprentissage

L'étape la plus importante dans l'utilisation d'un réseau de neurones est l'**apprentissage** des poids et seuils. Cependant, les choisir ou les calculer directement est impossible sur des problèmes complexes.

Il est donc nécessaire d'utiliser des algorithmes d'apprentissage. On peut les séparer dans trois grandes catégories.

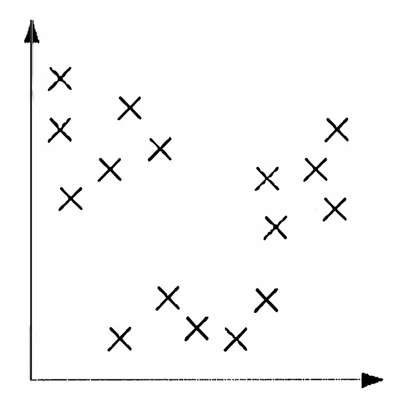
6.1 Apprentissage non supervisé

L'**apprentissage non supervisé** est la forme la moins courante d'apprentissage. En effet, dans cette forme d'apprentissage, il n'y a pas de résultat attendu. On utilise cette forme d'apprentissage pour faire du **clustering** : on a un ensemble de données, et on cherche à déterminer des classes de faits.

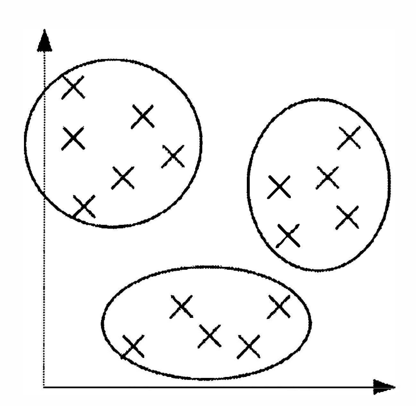
Par exemple, à partir d'une base de données de clients, on cherche à obtenir les différentes catégories, en fonction de leurs achats ou budgets. On ne sait pas a priori combien il y a de catégories ou ce qu'elles sont.

On va donc chercher à maximiser la cohérence des données à l'intérieur d'une même classe et à minimiser celle-ci entre les classes.

Imaginons que nous ayons l'ensemble de données suivant :



Si nous cherchions à déterminer des classes dans ces données, il serait possible de définir les trois suivantes :



De cette façon, on maximise bien la ressemblance entre les données d'une même classe (les points d'une classe sont proches) tout en minimisant les ressemblances entre les classes (elles sont éloignées entre elles).

Les algorithmes d'apprentissage non supervisé sortent du cadre de ce livre et ne sont donc pas présentés.

6.2 Apprentissage par renforcement

Dans l'**apprentissage par renforcement**, on indique à l'algorithme si la décision prise était bonne ou non. On a donc un retour global qui est fourni. Par contre, l'algorithme ne sait pas exactement ce qu'il aurait dû décider.

Remarque

C'est par exemple de cette façon que les animaux (et les humains) apprennent à marcher : on sait ce que l'on cherche à obtenir (la marche) mais pas comment l'obtenir (les muscles à utiliser, avec leur ordre). Le bébé essaie de marcher, et soit tombera (il a faux), soit il arrivera à faire un pas (il a juste). Il finira par comprendre par renforcement positif ou négatif ce qui lui permet de ne pas tomber, et deviendra meilleur, pour pouvoir arriver à courir ensuite.

Dans le cas des réseaux de neurones, on utilise souvent cette forme d'apprentissage quand on cherche à obtenir des comportements complexes faisant intervenir des suites de décisions. C'est par exemple le cas en robotique ou pour créer des adversaires intelligents dans les jeux vidéo. En effet, on cherche alors un programme qui prendra différentes décisions l'emmenant à une position où il est gagnant.

L'apprentissage non supervisé peut se faire grâce aux **métaheuristiques**. En effet, elles permettent d'optimiser des fonctions sans connaissances a priori. Cependant, la technique la plus employée est l'utilisation des **algorithmes génétiques**. Ils permettent, grâce à l'évolution, d'optimiser les poids et de trouver des stratégies gagnantes, sans informations particulières sur ce qui était attendu.

6.3 Apprentissage supervisé

L'**apprentissage supervisé** est sûrement le plus courant. Il est utilisé pour des tâches d'estimation, de prévision, de régression ou de classification.

6.3.1 Principe général

Dans l'apprentissage supervisé, un ensemble d'exemples est fourni à l'algorithme d'apprentissage. Celui-ci va comparer la sortie obtenue par le réseau avec la sortie attendue.

Les poids sont ensuite modifiés pour minimiser cette erreur, jusqu'à ce que les résultats soient satisfaisants pour tous les exemples fournis.

Cette approche est utilisée à chaque fois que les exemples sont présentés au réseau de neurones les uns à la suite des autres, sans liens entre eux.

C'est le cas en **estimation** où une valeur doit être calculée en fonction d'autres fournies : la consommation d'une maison en fonction de ses caractéristiques, la lettre dessinée en fonction des pixels noirs et blancs, la modification d'une valeur en bourse en fonction de son historique des dernières heures ou derniers jours, etc.

Remarque

On parle souvent de tâche de régression : il existe une fonction inconnue liant les entrées aux sorties que l'on cherche à approximer.

Selon le type de réseau choisi, les algorithmes d'apprentissage supervisés sont différents. Les trois principaux sont vus ici.

6.3.2 Descente de gradient

La **méthode de descente de gradient** ne s'applique qu'à des réseaux monocouches de type perceptron (avec une fonction heavyside). Les poids sont optimisés par plusieurs passes sur l'ensemble d'apprentissage.

Soit un réseau possédant X entrées, et N exemples. On note s_i la sortie obtenue sur le énième exemple, et y_i la sortie attendue. L'erreur commise sur un point s'exprime donc :

$$Erreur = y_i - s_i$$

Au début de chaque passe, on initialise à 0 les modifications à appliquer aux poids w_i. La variation est notée δw_i. À chaque exemple testé, on va modifier celle-ci de la manière suivante :

$$\delta w_i = \delta w_i + \tau . (y_i - s_i) . x_i$$

Remarque

Si la fonction d'activation est différente de la fonction heavyside, il est possible de généraliser cette méthode. Pour cela, il faut multiplier la modification appliquée par la dérivée de la fonction choisie.

Cette variation utilise donc l'erreur obtenue sur un exemple, mais aussi la valeur de l'entrée x_i et une constante τ appelée taux d'apprentissage. La modification d'un poids est donc plus importante si l'erreur est forte et/ou si l'entrée est importante.

Le **taux d'apprentissage** dépend lui du problème à résoudre : trop faible, il ralentit énormément la convergence. Trop grand, il peut empêcher de trouver la solution optimale. Ce taux sera généralement choisi fort au début de l'apprentissage puis sera réduit à chaque passe. Une variante consiste à diminuer le taux à chaque fois que l'erreur globale diminue, et de le remonter légèrement si l'erreur globale augmente.

Après avoir passé tous les exemples une fois, on applique la modification totale aux poids :

$$w_i = w_i + \delta w_i$$

Les modifications ne sont donc appliquées qu'après le test de tous les exemples.

On peut donc résumer cet algorithme par le pseudocode suivant :

```
Tantque critère d'arrêt non atteint
    Initialiser les dwi

    Pour chaque exemple :
           Calculer la sortie si
           Pour chaque poids :
                  dwi += taux * (yi - si) * xi
           FinPour
    FinPour

    Pour chaque poids :
           wi += dwi
    FinPour
```

```
    Si besoin, modification du taux
FinTantque
```

6.3.3 Algorithme de Widrow-Hoff

L'algorithme de descente du gradient converge, mais cependant il n'est pas très rapide. L'**algorithme de Widrow-Hoff** applique la même modification, mais au lieu de la faire après avoir vu tous les exemples, celle-ci est appliquée après chaque test.

De cette façon, l'algorithme converge plus vite sur la majorité des problèmes. Cependant, selon l'ordre et la valeur des exemples, l'algorithme peut en permanence osciller entre deux valeurs pour un poids.

Lorsqu'on applique cet algorithme, il faut faire attention à modifier l'ordre des exemples d'apprentissage régulièrement.

Son pseudocode est donc :

```
Tantque critère d'arrêt non atteint
    Pour chaque exemple :
           Calculer la sortie si
           Pour chaque poids :
                  wi += taux * (yi - si) * xi
           FinPour
    FinPour

    Si besoin, modification du taux
    Modification de l'ordre des exemples
FinTantque
```

Cet algorithme d'apprentissage est lui aussi limité à des réseaux de type perceptron (à une seule couche).

6.3.4 Rétropropagation

Les méthodes précédentes ne s'appliquent qu'aux perceptrons. Cependant, dans de nombreux cas, on utilisera des réseaux en couche (avec ici une fonction d'activation sigmoïde). Il existe un apprentissage possible : par **rétropropagation du gradient** (nommé "Backpropagation" en anglais).

On va donc corriger tout d'abord les poids entre les neurones de sortie et les neurones cachés, puis propager l'erreur en arrière (d'où le nom) et corriger les poids entre les neurones cachés et les entrées. Cette correction se fera exemple après exemple, et il faudra plusieurs passes (avec des ordres différents si possible) pour converger vers un optimum.

La première étape consiste donc à calculer la sortie de chaque neurone caché nommée o_i pour un exemple donné. On fait ensuite de même pour les neurones de sortie.

L'erreur commise est toujours :

$$Erreur = y_i - s_i$$

Avec y_i la sortie attendue et s_i celle obtenue.

L'étape suivante consiste à calculer les deltas sur les neurones de sortie. Pour cela, on calcule pour chacun la dérivée de la fonction d'activation multipliée par l'erreur, qui correspond à notre delta.

$$\delta_i = s_i \,.\, (1 - s_i).(y_i - s_i)$$

Il faut ensuite faire la même chose pour les neurones cachés, reliés chacun aux K neurones de sortie. Le calcul est alors :

$$\delta_i = o_i \,.\, (1 - o_i).\sum_k \delta_k * w_{i \to k}$$

En effet, ce calcul prend en compte la correction sur les neurones de sortie (δ_k) et le poids qui les relie. De plus, là encore, plus un poids est important et plus la correction à appliquer sera forte.

Lorsque tous les deltas sont calculés, les poids peuvent être modifiés en faisant le calcul suivant, où seule la dernière valeur change selon qu'il s'agit d'un neurone caché (on prend l'entrée) ou d'un neurone de sortie (on prend son entrée, c'est-à-dire la sortie du neurone caché) :

$$w_i = w_i + \tau \,.\, \delta_i \,.\, (x_i \; OU \; o_i)$$

Le pseudocode est donc le suivant :

```
Tantque critère d'arrêt non atteint
    Initialiser les d_i

    Pour chaque exemple :
          Calculer la sortie s_i

          Pour chaque poids des neurones de sortie :
                d_i = s_i * (1 - s_i) * (y_i - s_i)
          FinPour

          Pour chaque poids des neurones cachés :
                sum = 0
                Pour chaque lien vers le neurone de sortie k :
                      sum += d_k * w_i vers k
                FinPour
                d_i = o_i * (1 - o_i) * sum
          FinPour

          Pour chaque poids du réseau :
                Si lien vers neurone de sortie :
                      w_i += taux * d_i * o_i
                Sinon
                      w_i += taux * d_i * s_i
                FinSi
         FinPour
    FinPour


    Si besoin, modification du taux
FinTantque
```

6.4 Surapprentissage et généralisation

Lors de la phase d'apprentissage, il faut déterminer les critères d'arrêt. Ceux-ci permettent d'indiquer que le réseau a suffisamment bien appris les données fournies en exemple pour pouvoir ensuite être utilisé sur d'autres données.

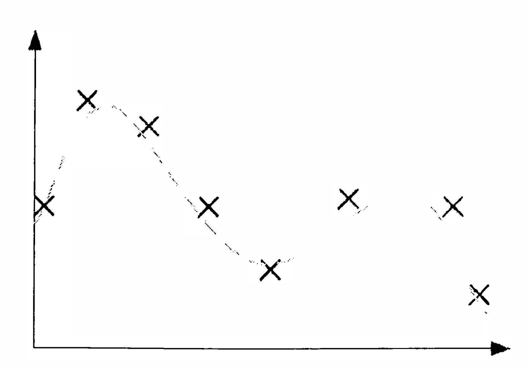
Cependant, il y a un fort risque de **surapprentissage** qu'il faut savoir détecter et contrer.

6.4.1 Reconnaître le surapprentissage

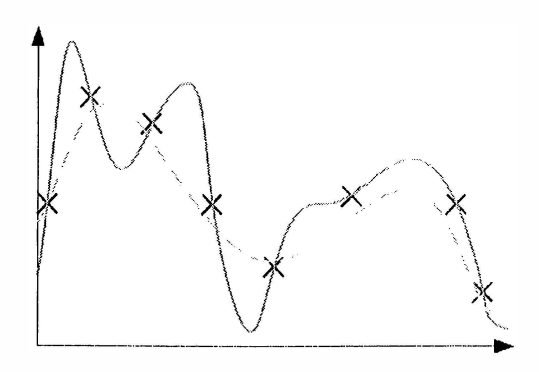
Le réseau apprend à partir des données qui lui sont fournies et va trouver une fonction globale permettant de limiter ses erreurs. Au départ, l'erreur sera forte, puis elle diminuera à chaque passage des données d'exemple et ajustement des poids.

Cependant, à partir d'un certain seuil, le réseau va apprendre les points fournis et perdre complètement en **généralisation**, surtout si les données fournies sont légèrement erronées : on a alors du **surapprentissage** (ou **over-fitting** en anglais).

Voici par exemple un problème simple, où il faut trouver la fonction généralisant les points donnés. Une bonne solution est proposée en pointillés.



Lorsque du surapprentissage apparaît, on peut alors se retrouver avec une fonction de ce type, qui passe par les points (l'erreur globale est donc nulle) mais perd complètement en généralisation :



Il est donc nécessaire non seulement d'évaluer la qualité de l'apprentissage, mais aussi la capacité de généralisation du réseau.

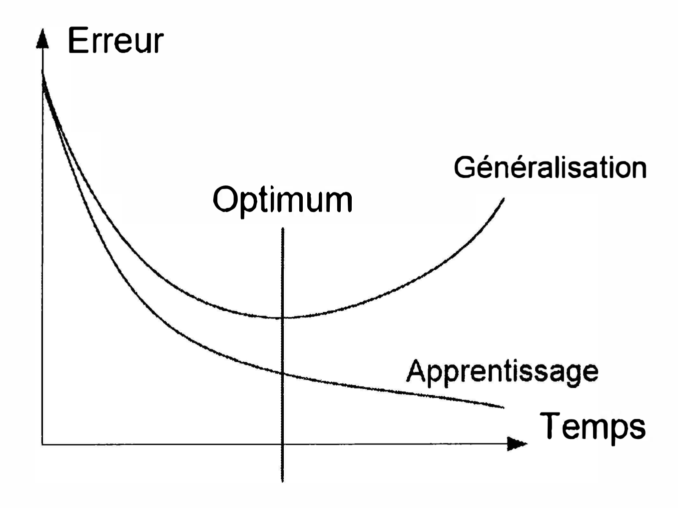
6.4.2 Création de sous-ensembles de données

Pour éviter le surapprentissage, ou au moins le détecter, nous allons séparer notre ensemble de données en trois sous-ensembles.

Le premier est l'**ensemble d'apprentissage**. C'est le plus important et il contient généralement 60 % des exemples. Il sert à l'algorithme d'apprentissage, pour adapter les poids et seuils du réseau.

Le deuxième ensemble, contenant environ 20 % des exemples, est l'**ensemble de généralisation**. À la fin de chaque passe, on teste l'erreur globale sur cet ensemble (qui n'a pas été utilisé pour changer les poids). Il nous indique à quel moment le surapprentissage apparaît.

En effet, si on trace au cours du temps l'erreur moyenne sur l'ensemble d'apprentissage et sur l'ensemble de validation, on obtient les courbes suivantes :



L'erreur sur l'ensemble d'apprentissage ne fait que baisser au cours du temps. Par contre, si dans un premier temps l'erreur sur la généralisation baisse, elle commence à augmenter lorsque le surapprentissage commence. C'est donc à ce moment-là qu'il faut arrêter l'apprentissage.

Le dernier ensemble est l'**ensemble de validation**. C'est lui qui permet de déterminer la qualité du réseau, pour comparer par exemple plusieurs architectures (comme un nombre de neurones cachés différent). Les exemples de cet ensemble ne seront vus par le réseau qu'une fois l'apprentissage terminé, ils n'interviennent donc pas du tout dans le processus.

7. Autres réseaux

Le perceptron et le réseau feed-forward sont les plus utilisés, mais il en existe de nombreux autres. Voici les trois principaux.

7.1 Réseaux de neurones récurrents

Dans un **réseau de neurones récurrent**, il existe non seulement des liens d'une couche vers les suivantes, mais aussi vers les couches précédentes.

De cette façon, les informations traitées à une étape peuvent être utilisées pour le traitement des entrées suivantes.

Cela permet d'avoir des suites de valeurs en sortie qui sont dépendantes, à la manière d'une série d'instructions pour un robot, ou un effet de mémorisation des pas de temps précédents.

Ces réseaux sont cependant très difficiles à ajuster. En effet, l'effet temporel complique les algorithmes d'apprentissage, et la rétropropagation ne peut pas fonctionner telle quelle.

7.2 Cartes de Kohonen

Les **cartes de Kohonen**, ou **cartes auto-adaptatives**, contiennent une grille de neurones. Au cours du temps, chacun va être associé à une zone de l'espace d'entrée, en se déplaçant à la surface de celui-ci.

Lorsque le système se stabilise, la répartition des neurones correspond à la topologie de l'espace. On peut donc ainsi faire une discrétisation de l'espace.

Ces cartes sont cependant peu utilisées dans des applications commerciales, à cause de leur complexité de mise en place.

7.3 Réseaux de Hopfield

Les **réseaux de Hopfield** sont des réseaux complètement connectés : chaque neurone est relié à tous les autres.

Lorsqu'on soumet une entrée au réseau, on ne modifie l'état que d'un neurone à la fois, jusqu'à la stabilisation du réseau. L'état stable est donc la "signature" de l'entrée.

L'apprentissage consiste à déterminer les poids de manière à ce que des entrées différentes produisent des états stables différents, mais que des entrées presque identiques conduisent au même état.

De cette façon, si des erreurs entachent légèrement une entrée, elle sera quand même reconnue par le réseau. On peut ainsi imaginer un système permettant la reconnaissance des lettres, même si celles-ci sont abîmées ou moins lisibles.

L'apprentissage dans ces réseaux se fait grâce à une variante de la **loi de Hebb**. Celle-ci indique qu'il faut renforcer la connexion entre deux neurones s'ils sont actifs en même temps, et diminuer le poids sinon.

8. Domaines d'application

Les réseaux de neurones sont utilisés dans de nombreux domaines. Ils sont une très bonne technique lorsque les critères suivants sont remplis :

- De nombreux exemples sont disponibles pour l'apprentissage, ou alors il est possible d'en créer facilement.
- Il n'existe pas de liens connus entre les entrées et les sorties exprimables par des fonctions.
- La sortie est plus importante que la façon de l'obtenir, les réseaux de neurones ne permettant pas d'avoir une explication sur le processus utilisé en interne.

8.1 Reconnaissance de patterns

La tâche la plus courante donnée à des réseaux de neurones est la **reconnaissance de patterns**.

Dans cette tâche, différents patterns sont présentés au réseau pendant l'apprentissage. Lorsque de nouveaux exemples doivent être classés, il peut alors reconnaître les motifs : il s'agit d'une tâche de **classification**.

C'est ainsi que les réseaux de neurones peuvent reconnaître des caractères manuscrits ou des formes. Des applications permettent de lire les plaques d'immatriculation dans une image même en présence de défauts d'éclairage ou sur la plaque en elle-même.

8.2 Estimation de fonctions

L'**estimation de fonctions** ou **régression**, consiste à donner une valeur numérique à partir d'entrées, en généralisant le lien existant entre celles-ci. Les entrées peuvent représenter des caractéristiques ou des séries temporelles selon les besoins.

Des applications en médecine sont ainsi possibles. Il existe des réseaux de neurones prenant en entrée des caractéristiques issues de radios de la main et du poignet et capables de déterminer la sévérité de l'arthrose, une maladie touchant les articulations.

Il est aussi possible en finance de déterminer la santé bancaire d'un individu pour lui associer un "score de crédit" indiquant si l'accord d'un crédit est risqué ou non. En bourse, ils permettent d'estimer les cours et/ou d'indiquer des valeurs qui semblent prometteuses.

8.3 Création de comportements

Si le réseau le permet, c'est tout un **comportement** qui pourra être défini par celui-ci. Les applications en robotique et dans l'industrie sont alors nombreuses.

Il est ainsi possible de contrôler un véhicule autonome, qui prendrait en entrée les informations sur l'environnement, et fournirait en sortie les ordres de déplacement.

D'autres études donnent la possibilité de contrôler des robots, leur permettant d'apprendre des trajectoires ou des séries d'actions.

Alstom, par exemple, les utilise aussi pour contrôler de manière plus efficace des processus industriels complexes.

9. Implémentation d'un MLP

Les MLP (*MultiLayer Perceptron*) sont des réseaux très utilisés. Ce sont des réseaux feed-forward, avec des neurones de type perceptron. La fonction d'agrégation est une somme pondérée, et la fonction d'activation une sigmoïde, ce qui permet un apprentissage par rétropropagation.

Le réseau codé ici possède une seule couche cachée. Le nombre de neurones des différentes couches comme le nombre d'entrées sont paramétrables.

Deux problèmes sont ensuite présentés :

- Le problème du XOR (ou exclusif) qui est simple à résoudre et permet de tester que les algorithmes fonctionnent.
- Le problème "Abalone", qui est de type régression et est très utilisé pour comparer des algorithmes d'apprentissage.

À l'exception du programme principal, qui est une application Windows de type console, le reste du code est compatible avec l'ensemble des plateformes .Net 4 et plus, Windows Store, Windows Phone et Silverlight, permettant de l'utiliser dans un maximum d'applications.

9.1 Points et ensembles de points

Les problèmes utilisés avec les réseaux de neurones nécessitent de nombreux points pour l'apprentissage. Il n'est donc pas concevable de les rentrer à la main dans le code.

On utilisera donc des fichiers textes, avec des tabulations comme séparateurs.

La première classe est **DataPoint**, correspondant à un exemple. Celui-ci contient un tableau de valeurs considérées comme des entrées et un tableau de valeurs de sortie. On rajoute deux propriétés permettant de récupérer ces tableaux qui ne sont pas modifiables.

Le début de la classe est le suivant :

```
using System;

internal class DataPoint
{
    double[] inputs;
    internal double[] Inputs
    {
        get
        {
            return inputs;
        }
    }

    double[] outputs;
    internal double[] Outputs
    {
        get
        {
            return outputs;
        }
    }

    // Constructeur ici
}
```

Le constructeur prend en paramètres la chaîne correspondant à la ligne du fichier texte, et le nombre de sorties des exemples (les valeurs sont soit des entrées soit des sorties). Le contenu est d'abord séparé sur les caractères correspondant à la touche tabulation (`'\t'`) grâce à la fonction `Split`. Ensuite, les entrées et les sorties sont transformées en nombres réels.

```
    internal DataPoint(string _str, int _outputNb)
    {
        string[] content = _str.Split(new char[] { '\t' },
StringSplitOptions.RemoveEmptyEntries);
```

```
            inputs = new double[content.Length - _outputNb];
            for (int i = 0; i < inputs.Length; i++)
            {
                  inputs[i] = Double.Parse(content[i]);
            }

            outputs = new double[_outputNb];
            for (int i = 0; i < _outputNb; i++)
            {
                  outputs[i] = Double.Parse(content[content.Length
 - _outputNb + i]);
            }
     }
```

La deuxième classe est **DataCollection**, qui correspond à l'ensemble des points d'exemple. Ceux-ci seront séparés en un ensemble d'apprentissage (`trainingPoints`) et un ensemble de généralisation (`generalisationPoints`) permettant de détecter le surapprentissage.

La base de la classe est la suivante :

```
using System;
using System.Collections.Generic;
using System.Linq;

internal class DataCollection
{
    DataPoint[] trainingPoints;
    DataPoint[] generalisationPoints;

    // Méthodes ici
}
```

Deux méthodes sont ajoutées, permettant de récupérer les points d'apprentissage et les points de généralisation.

```
   internal DataPoint[] Points()
   {
          return trainingPoints;
   }

   internal DataPoint[] GeneralisationPoints()
   {
```

```
        return generalisationPoints;
    }
```

La dernière méthode est le constructeur. Celui-ci prend en paramètres la chaîne correspondant à l'intégralité du fichier sous la forme d'un tableau (une ligne par case), le nombre de sorties, et le ratio correspondant aux points d'apprentissage. Par exemple 0.8 indique que 80 % des points sont utilisés pour l'apprentissage et donc 20 % pour la généralisation.

Les étapes sont les suivantes :

- Les points sont lus et créés un par un à partir de leur contenu.
- L'ensemble d'apprentissage est créé en prenant le nombre d'exemples requis. Ceux-ci sont choisis aléatoirement parmi les points restants.
- L'ensemble de généralisation est enfin créé à partir des exemples non encore sélectionnés.

```
    internal DataCollection(String [] _content, int _outputNb, double
_trainingRatio)
    {
        int nbLines = _content.Length;
        List<DataPoint> points = new List<DataPoint>();
        for (int i = 0; i < nbLines; i++)
        {
            points.Add(new DataPoint(_content[i],
_outputNb));
        }

        int nbTrainingPoints = (int) (_trainingRatio * nbLines);
        trainingPoints = new DataPoint[nbTrainingPoints];
        Random rand = new Random();
        for (int i = 0; i < nbTrainingPoints; i++)
        {
            int index = rand.Next (points.Count);
            trainingPoints[i] = points.ElementAt(index);
            points.RemoveAt(index);
        }

        generalisationPoints = points.ToArray();
    }
```

La lecture du fichier sera faite dans le programme principal, car celle-ci dépend de la plateforme choisie.

9.2 Neurone

La base de notre réseau est le neurone, codé dans la classe **Neuron**. Celui-ci possède trois attributs :

- Le tableau des poids le reliant aux différentes entrées et au biais (qui sera la dernière valeur).
- Le nombre d'entrées.
- La sortie, qui est un nombre réel et qui est enregistrée car elle sert à l'apprentissage et cela évitera d'avoir à la recalculer.

Celle-ci est d'ailleurs associée à une propriété.

```
using System;

class Neuron
{
    double[] weights;
    int nbInputs;

    double output;
    internal double Output
    {
            get
            {
                 return output;
            }
    }

    // Méthodes ici
}
```

Il n'est jamais utile de récupérer tous les poids, mais seulement un poids particulier. Pour cela, la méthode `Weight` renvoie celui correspondant à l'index demandé. De même, la méthode `AdjustWeight` modifie la valeur d'un poids donné, ce qui est nécessaire à l'apprentissage.

```
    internal double Weight(int index)
    {
           return weights[index];
    }
```

```
internal void AdjustWeight(int index, double value)
{
    weights[index] = value;
}
```

Le constructeur prend en paramètre le nombre d'entrées de cette cellule. En effet, chaque neurone, qu'il soit caché ou de sortie, possède un nombre de poids correspondant au nombre d'entrées plus le biais. Les poids sont initialisés aléatoirement entre -1 et +1. La sortie est initialisée à `NaN` (*Not A Number*) de manière à différencier le fait qu'elle soit calculée ou non.

```
internal Neuron(int _nbInputs)
{
    nbInputs = _nbInputs;
    output = Double.NaN;

    Random generator = new Random();

    weights = new double[(nbInputs + 1)];
    for (int i = 0; i < (nbInputs + 1); i++)
    {
        weights[i] = generator.NextDouble() * 2.0 - 1.0;
    }
}
```

La méthode d'évaluation prend en paramètre un tableau de valeurs. Si la sortie n'est pas encore calculée, alors on commence par faire la somme pondérée des poids multipliés par les entrées, puis on calcule la sortie en utilisant une sigmoïde comme fonction d'activation.

```
internal double Evaluate(double[] _inputs)
{
    if (output.Equals(Double.NaN))
    {
        double x = 0.0;

        for (int i = 0; i < nbInputs; i++)
        {
            x += _inputs[i] * weights[i];
        }
        x += weights[nbInputs];

        output = 1.0 / (1.0 + Math.Exp(-1.0 * x));
    }
```

```
        return output;
    }
```

Une deuxième méthode d'évaluation fonctionne à partir d'un `DataPoint`. Celle-ci se contente d'appeler la méthode précédente :

```
    internal double Evaluate(DataPoint _point)
    {
        double[] inputs = _point.Inputs;
        return Evaluate(inputs);
    }
```

La dernière méthode permet de réinitialiser la sortie, de manière à pouvoir traiter un nouvel exemple.

```
    internal void Clear()
    {
        output = Double.NaN;
    }
```

9.3 Réseau de neurones

Les neurones étant implémentés, il est possible de passer au réseau complet, dans une classe **`NeuralNetwork`**.

Celui-ci contient tout d'abord cinq attributs :

- Un tableau contenant les neurones cachés `hiddenNeurons`.
- Un tableau contenant les neurones de sortie `outputNeurons`.
- Trois entiers indiquant le nombre d'entrées, de neurones cachés et de sorties du réseau.

```
internal class NeuralNetwork
{
    Neuron[] hiddenNeurons;
    Neuron[] outputNeurons;
    int nbInputs;
    int nbHidden;
    int nbOutputs;

    // Reste de la classe ici
}
```

Le constructeur prend le nombre d'entrées, de neurones cachés et de sorties en paramètres. Les neurones sont alors créés (couche cachée et couche de sortie).

```
    internal NeuralNetwork(int _nbInputs, int _nbHidden,
int _nbOutput)
    {
            nbInputs = _nbInputs;
            nbHidden = _nbHidden;
            nbOutputs = _nbOutput;

            hiddenNeurons = new Neuron[nbHidden];
            for (int i = 0; i < nbHidden; i++)
            {
                    hiddenNeurons[i] = new Neuron(_nbInputs);
            }

            outputNeurons = new Neuron[nbOutputs];
            for (int i = 0; i < nbOutputs; i++)
            {
                    outputNeurons[i] = new Neuron(_nbHidden);
            }
    }
```

La méthode suivante est celle permettant d'évaluer la sortie pour un exemple donné. Pour cela, on commence par réinitialiser les sorties des différents neurones.

Ensuite, on calcule la sortie de chaque neurone caché, puis celle des neurones de sortie. La méthode se termine par le renvoi du tableau des sorties obtenues.

```
    internal double[] Evaluate(DataPoint _point)
    {
            foreach (Neuron n in hiddenNeurons)
            {
                    n.Clear();
            }
            foreach (Neuron n in outputNeurons)
            {
                    n.Clear();
            }

            double[] hiddenOutputs = new double[nbHidden];
            for (int i = 0; i < nbHidden; i++)
            {
                    hiddenOutputs[i] =
hiddenNeurons[i].Evaluate(_point);
```

```
            }
            double[] outputs = new double[nbOutputs];
            for (int outputNb = 0; outputNb < nbOutputs;
outputNb++)
            {
                   outputs[outputNb] =
outputNeurons[outputNb].Evaluate(hiddenOutputs);
            }

            return outputs;
    }
```

La dernière méthode, et la plus complexe, est celle permettant d'ajuster les poids du réseau grâce à l'algorithme de rétropropagation. En paramètres, on fournit le point testé et le taux d'apprentissage.

La première étape consiste à calculer les deltas pour chaque neurone de sortie en fonction de la formule vue précédemment. Ensuite, c'est le delta des neurones cachés qui est calculé. Enfin, on met à jour les poids des neurones de sortie (sans oublier leur biais) et ceux des neurones cachés (là encore avec leur biais).

```
    internal void AdjustWeights(DataPoint _point, double
_learningRate)
    {
            // Deltas pour les sorties
            double[] outputDeltas = new double[nbOutputs];
            for (int i = 0; i < nbOutputs; i++)
            {
                    double output = outputNeurons[i].Output;
                    double expectedOutput = _point.Outputs[i];
                    outputDeltas[i] = output * (1 - output) *
(expectedOutput - output);
            }

            // Deltas pour les neurones cachés
            double[] hiddenDeltas = new double[nbHidden];
            for (int i = 0; i < nbHidden; i++)
            {
                    double hiddenOutput = hiddenNeurons[i].Output;
                    double sum = 0.0;
                    for (int j = 0; j < nbOutputs; j++)
                    {
                            sum += outputDeltas[j] *
```

```
outputNeurons[j].Weight(i);
                 }
                 hiddenDeltas[i] = hiddenOutput * (1 - 
hiddenOutput) * sum;
          }

          double value;

          // Ajustement des poids des neurones de sortie
          for (int i = 0; i < nbOutputs; i++)
          {
                 Neuron outputNeuron = outputNeurons[i];
                 for (int j = 0; j < nbHidden; j++)
                 {
                        value = outputNeuron.Weight(j) + 
_learningRate * outputDeltas[i] * hiddenNeurons[j].Output;
                        outputNeuron.AdjustWeight(j, value);
                 }
                 // Et biais
                 value = outputNeuron.Weight(nbHidden) + 
_learningRate * outputDeltas[i] * 1.0;
                 outputNeuron.AdjustWeight(nbHidden, value);
                 }

          // Ajustement des poids des neurones cachés
          for (int i = 0; i < nbHidden; i++)
          {
                 Neuron hiddenNeuron = hiddenNeurons[i];
                 for (int j = 0; j < nbInputs; j++)
                 {
                        value = hiddenNeuron.Weight(j) + 
_learningRate * hiddenDeltas[i] * _point.Inputs[j];
                        hiddenNeuron.AdjustWeight(j, value);
                 }
                 // Et biais
                 value = hiddenNeuron.Weight(nbInputs) + 
_learningRate * hiddenDeltas[i] * 1.0;
                 hiddenNeuron.AdjustWeight(nbInputs, value);
          }
   }
```

Le réseau de neurones est alors complet, algorithme d'apprentissage compris.

9.4 IHM

Avant de coder le système gérant le réseau, on code une petite interface **IHM** qui nous permettra d'afficher différents messages. Selon l'application choisie, le programme principal pourra ensuite implémenter celle-ci pour faire des sorties console, des notifications, etc.

Elle ne contient donc qu'une seule méthode.

```
using System;

public interface IHM
{
    void PrintMsg(String _msg);
}
```

9.5 Système complet

La dernière classe générique **NeuralSystem** est celle gérant tout le réseau et la boucle d'apprentissage. Plusieurs critères d'arrêt sont utilisés :

- On voit apparaître du surapprentissage, c'est-à-dire que l'erreur sur l'ensemble de généralisation augmente au lieu de diminuer.
- On a atteint le résultat espéré, c'est-à-dire que l'erreur sur l'ensemble d'apprentissage est inférieure à un seuil.
- Le nombre maximal d'itérations a été atteint, et on s'arrête donc.

La classe `NeuralSystem` contient tout d'abord plusieurs attributs :

- Des données d'apprentissage `data`.
- Un réseau de neurones attaché `network`.
- Une `IHM` utilisée pour les affichages.
- Le taux d'apprentissage initial.
- L'erreur maximale.
- Le nombre maximal d'itérations.

Des valeurs par défaut sont proposées pour la configuration du réseau. La base est donc :

```
using System;

public class NeuralSystem
{
    DataCollection data;
    NeuralNetwork network;
    IHM ihm;

    // Configuration
    double learningRate = 0.3;
    double maxError = 0.005;
    int maxIterations = 10001;

    // Méthodes ici
}
```

La première méthode est le constructeur. Celui-ci prend de nombreux paramètres : le nombre d'entrées, de neurones cachés, de sorties, le contenu du fichier de données, le pourcentage d'exemples d'apprentissage par rapport à la généralisation et l'IHM.

```
   public NeuralSystem(int _nbInputs, int _nbHidden, int
_nbOutputs, String[] _data, double _trainingRatio, IHM _ihm)
   {
           data = new DataCollection(_data, _nbOutputs,
_trainingRatio);
           network = new NeuralNetwork(_nbInputs, _nbHidden,
_nbOutputs);
           ihm = _ihm;
   }
```

Les méthodes suivantes permettent de modifier la configuration en changeant le taux d'apprentissage (`LearningRate`), l'erreur maximale (`MaximumError`) et le nombre maximum d'itérations (`MaximumIterations`).

```
   public void LearningRate(double _rate)
   {
         learningRate = _rate;
   }

   public void MaximumError(double _error)
```

```
    {
          maxError = _error;
    }

    public void MaximumIterations(int _iterations)
    {
          maxIterations = _iterations;
    }
```

La dernière méthode est la méthode principale `Run`. On commence par initialiser les différentes variables. Ensuite, tant qu'un des critères d'arrêt n'a pas été atteint, on boucle :

- On met à jour les erreurs de l'itération précédente et on initialise les erreurs pour cette itération.
- Pour chaque point d'apprentissage, on calcule sa sortie et l'erreur commise, et on adapte les poids du réseau.
- Pour chaque point de généralisation, on calcule la sortie et l'erreur.
- Si l'erreur en généralisation a augmenté, on modifie le booléen `betterGeneralisation` pour arrêter l'apprentissage.
- Si l'erreur en apprentissage augmente, c'est que le taux d'apprentissage est trop fort, on le divise alors par deux.
- Et on termine par l'affichage des valeurs sur l'itération en cours (erreurs et taux).

```
    public void Run()
    {
          int i = 0;
          double totalError = Double.PositiveInfinity;
          double oldError = Double.PositiveInfinity;
          double totalGeneralisationError =
Double.PositiveInfinity;
          double oldGeneralisationError =
Double.PositiveInfinity;
          Boolean betterGeneralisation = true;

          while (i < maxIterations && totalError > maxError &&
betterGeneralisation)
          {
                oldError = totalError;
                totalError = 0;
                oldGeneralisationError = totalGeneralisationError;
```

```
                totalGeneralisationError = 0;

                  // Évaluation
                  foreach (DataPoint point in data.Points())
                  {
                        double[] outputs = network.Evaluate(point);
                        for (int outNb = 0; outNb <
outputs.Length; outNb++)
                        {
                              double error = point.Outputs[outNb]
- outputs[outNb];
                              totalError += (error * error);
                        }

                       // Calcul des nouveaux poids par rétropropagation
                        network.AdjustWeights(point,
learningRate);
                  }

                  // Généralisation
                  foreach (DataPoint point in
data.GeneralisationPoints())
                  {
                        double[] outputs = network.Evaluate(point);
                        for (int outNb = 0; outNb <
outputs.Length; outNb++)
                         {
                              double error = point.Outputs[outNb]
- outputs[outNb];
                              totalGeneralisationError += (error *
error);
                        }
                }
                if (totalGeneralisationError >
oldGeneralisationError)
                {
                        betterGeneralisation = false;
                }

                // Changer le taux
                if (totalError >= oldError)
                {
                        learningRate = learningRate / 2.0;
                }

                // Information et incrément
                ihm.PrintMsg("Iteration n°" + i + " - Total
error : " + totalError + " - Gener Error : " +
```

```
totalGeneralisationError + " - Rate : " + learningRate + " - Mean :
" + String.Format("{0:0.00}",
Math.Sqrt(totalError/data.Points().Length),"%2"));
                i++;
            }
    }
```

Tout le système est prêt, apprentissage et boucle principale compris.

9.6 Programme principal

La dernière étape consiste à créer le programme principal **MainProgram**. Celui-ci implémente l'interface IHM, et il possède donc une méthode `PrintMsg`. Il possède aussi une méthode `ReadFile` qui se contente de récupérer toutes les lignes d'un fichier indiqué.

Enfin, la méthode `Main` instancie la classe et appelle sa méthode `Run` qui est définie ultérieurement en fonction du problème choisi.

Le code est donc :

```
using NeuralNetworkPCL;
using System;
using System.IO;
using System.Linq;

class MainProgram : IHM
{
    static void Main(string[] args)
    {
        MainProgram prog = new MainProgram();
        prog.Run();
    }

    private void Run()
    {
        // À compléter plus tard
    }

    private String[] ReadFile(String _filename, bool _removeFirstLine)
    {
        String[] content = File.ReadAllLines(@_filename);
        if (_removeFirstLine)
        {
```

```
                content = content.Skip(1).ToArray();
            }
            return content;
        }

        public void PrintMsg(string _msg)
        {
            Console.Out.WriteLine(_msg);
        }
    }
```

9.7 Applications

9.7.1 Application au XOR

Le premier problème que nous utilisons est celui de l'opérateur booléen **XOR** nommé en français "ou exclusif". Si nous remplaçons vrai par 1 et faux par 0, voici la table de vérité du XOR :

X	Y	X XOR Y
0	0	0
0	1	1
1	0	1
1	1	0

Remarque

Le XOR indique qu'une des deux valeurs vaut "vrai", mais pas les deux en même temps.

Contrairement à d'autres opérateurs booléens (comme le "et" et le "ou"), celui-ci n'est pas linéairement séparable. Il est donc un bon test pour un réseau de neurones.

La première étape consiste à créer le fichier contenant les exemples, ici au nombre de quatre. Vu leur petit nombre, il n'y aura pas d'ensemble de généralisation.

Le fichier xor.txt contient donc le texte suivant :

```
X    Y        XOR
0    0        0
0    1        1
1    0        1
1    1        0
```

Pour apprendre ce problème, on définit la méthode `Run` de la classe **`MainProgram`**. Celle-ci consiste donc à lire le fichier, puis à créer le système. On choisit deux neurones dans la couche cachée. Enfin, on lance l'apprentissage.

```
    private void Run()
    {
           // Problème du OU Exclusif (XOR)
           String[] content = ReadFile("xor.txt", true);
           NeuralSystem system = new NeuralSystem(2, 2, 1,
content, 1.0, this);

           system.Run();

           while (true) ;
    }
```

Remarque

Le fichier xor.txt doit être ajouté à la solution. Dans ses propriétés, il faut bien vérifier qu'il est ajouté en tant que contenu, et qu'il sera copié dans la solution compilée ("copier si plus récent").

Avec les paramètres choisis, le réseau de neurones converge et s'arrête lorsque l'erreur résiduelle de 0.005 est atteinte. À ce moment-là, la sortie est en moyenne de 0.97 pour les sorties vraies et 0.03 pour les sorties fausses. L'erreur est donc minime, et si on arrondit les sorties (car il s'agit de booléens) celle-ci devient nulle.

Cependant, quelques fois, il reste bloqué dans un optimum local et n'arrive plus à en sortir. C'est pourquoi il est intéressant de relancer plusieurs fois l'apprentissage pour garder les meilleurs réseaux.

9.7.2 Application à Abalone

Le deuxième problème est plus complexe. Il est défini sur l'UCI (*University of California, Irvine*) Machine Learning Repository. Il s'agit d'une banque de données contenant de nombreux datasets qui peuvent ainsi être utilisés pour faire des tests dans le domaine de l'apprentissage automatique.

Abalone est présenté à l'adresse suivante :

```
https://archive.ics.uci.edu/ml/datasets/Abalone
```

Cet ensemble, déposé en 1995, propose de déterminer l'âge d'ormeaux (il s'agit de coquillages nommés abalone en anglais) en fonction de caractéristiques physiques. Pour cela, on dispose de huit données :

- Le sexe de l'animal, au choix parmi mâle, femelle ou enfant.
- La longueur de la coquille la plus longue.
- Le diamètre mesuré perpendiculairement à la longueur.
- La hauteur.
- Le poids total du coquillage.
- Le poids de l'animal (sans la coquille).
- Le poids des viscères (donc après l'avoir saigné).
- Le poids de la coquille sèche.

La sortie recherchée est le nombre d'anneaux de la coquille, qui correspond à l'âge de l'animal (en ajoutant 1.5). Ceux-ci ont tous entre 1 et 29 ans.

4177 exemples sont fournis, ce qui permet de créer un ensemble d'apprentissage contenant 80 % des données (soit 3341 animaux) et 20 % pour tester la généralisation (soit 836).

Le fichier .csv contenant les données est disponible sur le site. Cependant, pour pouvoir l'utiliser avec notre réseau de neurones, nous remplaçons la première donnée (le sexe) par trois données booléennes : est-ce un mâle, une femelle ou un enfant ? On a donc remplacé notre donnée textuelle par trois entrées contenant 0 (faux) ou 1 (vrai), ce qui porte à 10 le nombre de variables en entrée.

La sortie du réseau étant le résultat d'une sigmoïde, on ne peut qu'obtenir une sortie comprise entre 0 et 1. Comme l'âge est ici compris entre 1 et 29, on va normaliser celui-ci. Pour cela, on se contentera de diviser l'âge par 30.

Le fichier obtenu après les modifications est téléchargeable sur le site de l'éditeur, dans la solution Visual Studio.

Pour résoudre ce problème, on choisit quatre neurones cachés, et un taux d'apprentissage de 0.1. La méthode Run de la classe **MainProgram** devient donc :

```
    private void Run()
    {
           // Problème Abalone
           String[] content = ReadFile("abalone_norm.txt", false);
           NeuralSystem system = new NeuralSystem(10, 4, 1,
content, 0.8, this);
           system.LearningRate(0.1);

           system.Run();

           while (true) ;
    }
```

Le fichier "abalone_norm.txt" doit lui aussi être ajouté à la solution en tant que contenu et être copié dans le répertoire du code compilé.

À des fins statistiques, 10 simulations ont été lancées. Lors de la première génération, l'erreur quadratique cumulée est de 37.36 en moyenne, et sur l'ensemble de généralisation de 7.88.

Lorsque l'apprentissage s'arrête, elle n'est plus que de 15.98, et une erreur de généralisation de 3.88. L'erreur a donc été divisée par plus de 2 : l'apprentissage a bien permis d'améliorer les résultats. Cela correspond à une erreur moyenne de moins de 2 ans sur chaque ormeau, qui pour rappel a entre 1 et 29 ans.

De plus, dans plus de la moitié des cas, l'apprentissage s'est arrêté car il avait atteint les 10000 itérations. Cela signifie que si cette limite était repoussée, l'apprentissage pourrait encore s'améliorer. Le nombre de neurones cachés n'a pas non plus été optimisé, une étude serait nécessaire pour connaître le nombre optimal.

Bien que non optimisé, on voit cependant que les réseaux de neurones arrivent à apprendre à partir de données, et à fournir des résultats de qualité.

9.7.3 Améliorations possibles

Comme on l'a vu précédemment, la première amélioration consisterait à optimiser les différents paramètres d'apprentissage :

- Le nombre de neurones cachés.
- Le taux d'apprentissage.
- Le nombre maximal d'itérations.

De plus, les résultats pourraient être améliorés en modifiant la stratégie de modification du taux d'apprentissage, ou encore en modifiant l'ordre des exemples présentés au réseau à chaque itération.

Les résultats présentés ici le sont donc surtout pour illustrer le fonctionnement de ces réseaux, sans recherche de résultats optimaux.

10. Synthèse du chapitre

Les **réseaux de neurones** ont été inspirés du fonctionnement du cerveau des êtres vivants. En effet, de simples cellules ne faisant que transmettre des impulsions électriques en fonction des entrées reçues permettent l'ensemble des comportements et des réflexions. Leur puissance émerge du nombre des cellules grises et de leurs connexions.

Le neurone artificiel, dit **neurone formel**, combine une **fonction d'agrégation** permettant d'obtenir une unique valeur à partir de l'ensemble des entrées, des poids du neurone et de son biais, et une **fonction d'activation**, permettant d'obtenir sa sortie.

La fonction d'agrégation est généralement une somme pondérée. La fonction d'activation est plus variée, mais correspond à la fonction signe (ou heavyside), la fonction sigmoïde ou la fonction gaussienne dans la majorité des cas.

Les réseaux à une seule couche sont cependant limités aux problèmes **linéairement séparables**, c'est pourquoi les réseaux les plus utilisés sont de type **feed-forward** : les entrées passent dans une première couche de neurones, entièrement connectée à la suivante et ainsi de suite jusqu'à la couche de sortie.

Quel que soit le type choisi, il faut cependant ajuster les poids et les seuils pour un problème donné. Cette étape **d'apprentissage** est complexe et quasiment impossible à faire "à la main". De nombreux algorithmes d'apprentissage existent donc, qu'ils soient **non supervisés**, **par renforcement** ou **supervisés**.

Dans ce dernier cas et pour les réseaux à une seule couche de type perceptron, on peut utiliser la **descente de gradient** ou l'**algorithme de Widrow-Hoff**. Pour les réseaux de type feed-forward, on applique généralement l'**algorithme de rétropropagation**. Celui-ci consiste à propager l'erreur sur la couche de sortie aux couches cachées, l'une après l'autre, et à corriger les poids de chaque couche pour diminuer l'erreur totale.

La difficulté consiste cependant à éviter le **surapprentissage**. Pour cela, il faut comparer l'évolution de l'erreur totale sur la partie des données présentée pour l'apprentissage et sur l'ensemble de généralisation, qui n'est pas utilisé. Dès que l'erreur en généralisation augmente, il faut arrêter l'apprentissage.

Ces réseaux sont utilisés dans de nombreux domaines, car ils donnent de bons résultats, pour un effort de mise en place assez faible. De plus, ils permettent de résoudre des problèmes trop complexes pour des techniques plus classiques.

Une **implémentation** possible d'un réseau feed-forward a été proposée dans ce chapitre, et appliquée au problème simple du XOR et sur celui plus complexe nommé Abalone, consistant à déterminer l'âge de coquillages à partir de données physiques. Dans les deux cas, le réseau donne de bons résultats, sans avoir pour autant été optimisé complètement.

Bibliographie

Ce livre présente différentes techniques d'intelligence artificielle, sans rentrer dans tous les détails et toutes les possibilités. Cette bibliographie permet donc aux lecteurs le souhaitant d'approfondir une des techniques en particulier.

Systèmes experts, un nouvel outil pour l'aide à la décision, J.-M. Karkan et G. Tjoen, Éd. Elsevier-Masson, 1993

Il s'agit de la référence sur les systèmes experts, même si ce livre date déjà de quelques années. Il présente la théorie sur la représentation des connaissances, les moteurs d'inférences, les bases de connaissances, la construction d'un système expert, etc. La création d'un système expert est aussi expliquée.

Prolog, tout de suite !, P. Blackburn, J. Bos et K. Streignitz, Éd. College Publications, 2007

Prolog est un langage de programmation logique très particulier à prendre en main pour des développeurs, car il s'agit d'un fonctionnement éloigné de la programmation orientée objet. Ce livre présente donc le langage à travers de nombreux exemples et exercices.

La logique floue et ses applications, B Bouchon-Meunier, Éd. Addison-Wesley, 1990

Ce livre est un des rares portant sur la logique floue. De plus, il est écrit par une spécialiste internationale du domaine et préfacé par L. Zadeh, le fondateur de la logique floue. Les concepts et principes sont présentés. Les applications ont une part très importante.

Théorie des graphes et applications, avec exercices et problèmes, J.-C. Fournier, Éd. Lavoisier, 2011

Beaucoup d'algorithmes liés à la théorie des graphes sont ici présentés, dont les algorithmes de recherche de chemins optimaux, ou l'optimisation combinatoire. De plus, des exercices sont proposés à chaque chapitre.

Algorithmes génétiques, exploration, optimisation et apprentissage automatique, D. Goldberg, traduit par V. Corruble, Éd. Addison-Wesley, 1996

David Goldberg est un des fondateurs des algorithmes génétiques, et aussi un des plus grands spécialistes du domaine actuellement. Il partage sa notoriété avec J. Holland, auteur de la préface. Ce livre est considéré comme une référence sur les algorithmes génétiques. Il présente les principes mathématiques et des applications, ainsi qu'une implémentation en PASCAL.

Métaheuristiques, **Recuit simulé**, **recherche avec tabous**, **recherche à voisinages variables**, **méthode GRASP**, **algorithmes évolutionnaires**, **fourmis artificielles**, **essaims particulaires et autres méthodes d'optimisation**, sous la direction de P. Siarry, Éd. Eyrolles, 2014

Ce livre présente les principales métaheuristiques dont nous avons parlé mais aussi de nombreuses autres, ou des variantes. Une dizaine d'auteurs, tous spécialistes dans leur domaine, présentent ainsi les algorithmes et applications possibles.

Les systèmes multi-agents, vers une intelligence collective, J. Ferber, Éd. InterEditions, 1997

Dans ce livre, un des rares sur les systèmes multi-agents, sont présentés les principes et différents concepts liés, sans se limiter au point de vue informatique. D'autres disciplines sont ainsi utilisées pour les comprendre, comme la sociologie ou la biologie.

Les réseaux de neurones, présentation et applications, P. Borne, M. Benrejeb et J. Haggège, Éd. Technip, 2007

Dans cet ouvrage, les fondamentaux des réseaux de neurones ainsi que les algorithmes d'apprentissage sont présentés. Des variantes aux réseaux classiques sont présentées, comme les réseaux de Hopfield ou les réseaux neuro-flous combinant les neurones et la logique floue.

Sitographie

1. Pourquoi une sitographie ?

Cette sitographie présente différents liens vers des applications en intelligence artificielle. Elle permet d'avoir un aperçu de l'utilisation réelle qui est faite de telle ou telle technique, chacune ayant sa section.

Les articles pointés sont en français ou en anglais. Ces derniers sont indiqués par la mention [EN] à la suite de leur titre.

Cette liste est bien sûr loin d'être exhaustive mais présente des applications très différentes.

2. Systèmes experts

Applications of Artificial Intelligence for Organic Chemistry [EN], R. Lindsay, B. Buchanan, E. Feigenbaum, J. Lederberg, 1980 :

http://profiles.nlm.nih.gov/ps/access/BBALAF.pdf

Ce PDF présente le projet Dendral, qui est le premier gros système expert créé dans les années 60. Il permettait de reconnaître les composants chimiques en fonction de leurs caractéristiques.

MYCIN : A Quick Case Study [EN], A. Cawsey, 1994 :

http://cinuresearch.tripod.com/ai/www-cee-hw-ac-uk/_alison/ai3notes/section2_5_5.html

MYCIN est un autre système expert fondateur (développé dans les années 70) et reconnu mondialement. Il permettait d'identifier les principales maladies du sang et proposait un traitement. Il s'agit ici d'une courte discussion sur ses points forts et faibles.

Clinical decision support systems (CDSSs) [EN], Open Clinical, 2006 :

http://www.openclinical.org/dss.html

Cette page présente plusieurs systèmes experts utilisés en médecine ainsi que leur fonctionnement général et les principaux travaux publiés.

Machine learning for an expert system to predict preterm birth risk [EN], Journal of the American Medical Informatics Association, L Woolery, J Grzymala-Busse, 1994 :

http://www.ncbi.nlm.nih.gov/pmc/articles/PMC116227/

Il s'agit d'un article scientifique publié dans la revue Journal of the American Medical Informatics Association présentant un système expert permettant d'estimer le risque de naissances prématurées.

An Expert System for Car Failure Diagnosis [EN], Proceedings of World Academy of Science, Engineering and Technology, volume 7, A. Al-Taani, 2005 :

http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.99.3377

Cet article scientifique, téléchargeable en PDF est lui aussi issu d'une revue (Proceedings of World Academy of Science, Engineering and Technology, volume 7, 2005). Il propose un système expert permettant de détecter les pannes sur des voitures. Les difficultés rencontrées lors de la mise en œuvre sont aussi expliquées.

Expert System in Real World Applications [EN], Generation 5, K. Wai, A. Abdul Rahman, M. Zaiyadi, A. Aziz, 2005 :

http://www.generation5.org/content/2005/Expert_System.asp

Il s'agit d'un article publié sur le site "Generation 5" et qui donne une vue d'ensemble d'applications des systèmes experts, en particulier en agriculture, dans l'éducation, en gestion de l'environnement et en médecine. Il s'agit donc d'un très bon point d'entrée aux principales applications de ces domaines.

Clé de détermination générale, Microcox.net, 2006 :

http://microcox.pagesperso-orange.fr/clef_generale.htm

Cette clé de détermination permet, à partir de l'application de règles simples, de reconnaître les principaux insectes (ou au moins leur famille). Pour cela, il suffit de répondre aux différentes questions qui s'enchaînent.

3. Logique floue

Fiche produit "Washing Machines - LG T8018EEP5" [EN], LG, 2014 :

http://www.lg.com/in/washing-machines/
lg-T8018AEEP5-top-loading-washing-machine

Cette fiche produit pour une machine à laver de marque LG met en avant l'utilisation d'un contrôleur flou pour le choix de la quantité d'eau et le temps de lavage. La logique floue est un argument de vente.

Brevet "Fuzzy logic control for an electric clothes dryer" [EN], déposé par Whirlpool Corporation, 2001 :

https://www.google.com/patents/US6446357

Le brevet, déposé par la marque d'électroménager Whirlpool, indique le fonctionnement d'un sèche-linge utilisant de la logique floue. Ce contrôleur permet de choisir le temps de séchage en fonction de la charge et de l'humidité des vêtements.

Fuzzy Logic in Automotive Engineering [EN], Circuit Cellar Ink, Issue 88, C. von Altrock, 1997 :

http://www.cs.dartmouth.edu/~spl/Academic/RealTimeSystems/
Spring2002/Labs/FuzzyControl/FuzzyLogicInAutomativeEngineering.pdf

Cette édition du journal est consacrée aux applications de la logique floue dans l'automobile. Les avancées depuis ont été très importantes, mais les contrôleurs alors utilisés le sont toujours, et dans toutes les grandes marques.

Fuzzy logic method and apparatus for battery state of health determination [EN], USPTO Patent Database, déposé par Cadex, 2001 :

http://patft.uspto.gov/netacgi/nph-Parser?Sect1=PTO1&Sect2=HITOFF&d=PALL&p=1&u=/netahtml/PTO/srchnum.htm&r=1&f=G&l=50&s1=7,072,871.PN.&OS=PN/7,072,871&RS=PN/7,072,871

Ce brevet présente l'utilisation de logique floue pour déterminer l'état d'une pile ou d'une batterie à partir de paramètres électrochimiques.

The use of Fuzzy Logic for Artificial Intelligence in Games [EN], M. Pirovano, 2012 :

http://homes.di.unimi.it/~pirovano/pdf/fuzzy_ai_in_games.pdf

Dans cet article, l'auteur présente de nombreuses utilisations de la logique floue dans les jeux vidéo après un rappel de ce qu'est l'intelligence artificielle et plus particulièrement les concepts de logique floue. De nombreux liens sont aussi disponibles vers d'autres articles pour approfondir ce sujet.

A Computer Vision System for Color Grading Wood Boards Using Fuzzy Logic [EN], IEEE International Symposium on Industrial Electronics, J. Faria, T. Martins, M. Ferreira, C. Santos, 2008 :

http://repositorium.sdum.uminho.pt/bitstream/1822/16874/1/C24-%20A%20Computer%20Vision%20System%20for%20Color%20Grading%20Wood%20Boards%20Using%20Fuzzy%20Logic_2.pdf

Cet article, publié lors d'une conférence, permet de voir une application plus atypique de la logique floue. En effet, elle est ici utilisée pour déterminer la couleur d'un bois, pour pouvoir grouper les planches par teinte (par exemple pour en faire des meubles dans les mêmes teintes).

Leaf Disease Grading by Machine Vision and Fuzzy Logic [EN], International Journal of Computer Technology and Applications Vol 2 (5), S. Sannakki, V. Rajpurohit, V. Nargund, A. Kumar R, P. Yallur, 2011 :

http://ijcta.com/documents/volumes/vol2issue5/ijcta2011020576.pdf

La logique floue est ici appliquée à la recherche de maladies à partir d'images des feuilles de plantes. En plus, le système permet de déterminer la gravité actuelle de la maladie.

4. Algorithmes génétiques

Algorithmes génétiques appliqués à la gestion du trafic aérien, Journal sur l'enseignement des sciences et technologies de l'information et des systèmes 2, Hors-Série 1, N. Durand, J.-B. Gotteland, 2003 :

http://hal-enac.archives-ouvertes.fr/hal-00991623

Deux applications des algorithmes génétiques au contrôle aérien sont proposées dans cet article et comparées à des méthodes traditionnelles. De plus, toutes les étapes de modélisation puis d'évolution sont présentées.

Site web du laboratoire "Notredame's Lab Comparative Bioinformatics" [EN], C. Notredame :

http://www.tcoffee.org/homepage.html

Ce laboratoire situé à Barcelone dans le centre de régulation génomique utilise de manière importante les algorithmes génétiques pour résoudre des problèmes biologiques et plus particulièrement dans l'analyse et l'alignement de séquences ARN. Les différents projets y sont présentés.

Staples' Evolution [EN], BloombergBusinessweek, Innovation & Design, J. Scanlon, 2008 :

http://www.businessweek.com/stories/2008-12-29/staples-evolutionbusinessweek-business-news-stock-market-and-financial-advice

Cet article, paru sur un journal économique, étudie la stratégie marketing de Staples. En effet, cette entreprise a utilisé un algorithme génétique pour relancer sa marque en améliorant le packaging de ses ramettes de papier.

Algorithmes génétiques & art évolutionnaire, K. Auguste, 2010 :

http://www.palkeo.com/projets/algorithmes_genetiques/index.html

Alors étudiant, K. Auguste s'est amusé à créer des images colorées à partir d'un réseau de neurones dont les poids évoluaient avec un algorithme génétique. Le mélange des deux techniques est souvent une réussite, et l'utilisation artistique est plutôt rare.

A (R)evolution in Crime-fighting [EN], Forensic Magazine, C. Stockdale, 2008 :

http://www.forensicmag.com/articles/2008/06/revolution-crime-fighting

Cet article présente la difficulté de créer des portraits robots à partir des souvenirs des témoins. Cette phase peut être simplifiée par un algorithme génétique qui ferait évoluer ces portraits, en laissant le choix à l'utilisateur qui aurait à choisir le portrait "le plus proche" de celui dont il se souvient.

5. Recherche de chemins

Path-Finding Algorithm Applications for Route-Searching in Different Areas of Computer Graphics [EN], New Frontiers in Graph Theory, chap. 8, C. Szabó, B. Sobota, 2012 :

http://cdn.intechopen.com/pdfs-wm/29857.pdf

Il s'agit d'un chapitre extrait d'un livre plus complet sur la théorie des graphes. Les auteurs s'intéressent ici aux algorithmes de recherche de chemins et leurs applications en imagerie informatique.

Robots and Video Games Have Similar Approaches to A.I. Pathfinding [EN], Examiner.com, P. Vaterlaus, 2011 :

http://www.examiner.com/article/robots-and-video-games-have-similar-approaches-to-a-i-pathfinding

Ce petit article compare les besoins en recherche de chemins des jeux vidéo modernes et de la robotique, les deux présentant les mêmes caractéristiques, ainsi que différentes approches possibles, sans entrer dans les détails algorithmiques.

The Bellman-Ford routing algorithm [EN], FortiOS Online Handbook, 2014 :

http://docs-legacy.fortinet.com/fos50hlp/50/index.html#page/FortiOS%205.0%20Help/routing_rip.023.16.html

Il s'agit d'une partie du livre sur FortiOS qui présente les différents algorithmes de routage, et plus précisément ici l'implémentation de l'algorithme de Bellman-Ford pour le routage RIP.

OSPF Background and concepts [EN], FortiOS Online Handbook, 2014 :

http://docs-legacy.fortinet.com/fos50hlp/50/index.html#page/FortiOS%25205.0%2520Help/routing_ospf.025.02.html

Cette partie, issue du même livre que le lien précédent, explique quant à elle le protocole OSPF, qui remplace RIP. Au lieu d'utiliser Bellman-Ford pour la recherche de chemins, c'est Dijkstra qui est implémenté.

Les secrets d'une machine surpuissante. L'ordinateur Deep Blue joue aux échecs avec une "mémoire" alimentée par l'homme, Libération, D. Leglu, 1997 :

http://www.liberation.fr/evenement/1997/05/13/les-secrets-d-une-machine-surpuissante-l-ordinateur-deep-blue-joue-aux-echecs-avec-une-memoire-alime_205645

Dans cet article est présentée Deep Blue, la machine capable de battre les meilleurs joueurs d'échecs au monde (dont Kasparov).

6. Métaheuristiques

A Comparative Study on Meta Heuristic Algorithms for Solving Multilevel Lot-Sizing Problems [EN], Recent Advances on Meta-Heuristics and Their Application to Real Scenarios, I. Kaku, Y. Xiao, Y. Han, 2013 :

http://www.intechopen.com/books/recent-advances-on-meta-heuristics-and-their-application-to-real-scenarios/a-comparative-study-on-meta-heuristic-algorithms-for-solving-multilevel-lot-sizing-problems

Il s'agit d'un chapitre de livre dédié à une application industrielle des métaheuristiques, et disponible gratuitement. Le problème principal est celui du choix des quantités de chaque composant à produire. Plusieurs algorithmes sont comparés.

A Two-Step Optimization Method for Dynamic Weapon Target Assignment Problem [EN], Recent Advances on Meta-Heuristics and Their Application to Real Scenarios, C. Leboucher, H.-S. Shin, P. Siarry, R. Chelouah, S. Le Ménec, A. Tsourdos, 2013 :

http://www.intechopen.com/books/recent-advances-on-meta-heuristics-and-their-application-to-real-scenarios/a-two-step-optimisation-method-for-dynamic-weapon-target-assignment-problem

Les militaires trouvent aussi de nombreuses applications aux métaheuristiques. Ce chapitre est issu du même livre que précédemment mais dédié à une application militaire, consistant à savoir comment assigner les moyens de défense (ou d'attaque) en fonction des différentes menaces.

Metaheuristics and applications to optimization problems in telecommunications [EN], Handbook of optimization in telecommunications, S. Martins, C. Ribeiro, 2006 :

http://www-di.inf.puc-rio.br/~celso/artigos/metahot.ps

Les télécommunications sont un domaine demandant de nombreuses optimisations. Ce livre y est d'ailleurs dédié. Parmi toutes les techniques possibles, les métaheuristiques sont en bonne place, faisant l'objet de tout le chapitre 1 ici indiqué.

7. Systèmes multi-agents

MASSIVE [EN], site web du logiciel, 2011 :

http://www.massivesoftware.com/

Les films et jeux vidéo sont gourmands en systèmes multi-agents, mais ils peuvent aussi être utilisés en éducation, architecture ou pour des simulations. MASSIVE est un logiciel très utilisé permettant de simuler des foules. Les pages de références sont impressionnantes.

Ant Colony Optimization - Techniques and applications [EN], edited by H. Barbosa, 2013 :

http://www.intechopen.com/books/ant-colony-optimization-techniques-and-applications

Il s'agit d'un livre complet uniquement dédié aux algorithmes à base de fourmis, et plus particulièrement à leurs applications dans différents domaines, comme la logistique, ou à des variantes/extensions.

An ant colony optimization algorithm for job shop scheduling problem [EN], E. Flórez, W. Gómez, L. Bautista, 2013 :

http://arxiv.org/abs/1309.5110

Cet article s'intéresse à l'utilisation d'un algorithme à colonie de fourmis dans le domaine de la planification d'opérations. Une variante est d'ailleurs utilisée dans ces travaux.

Train scheduling using ant colony optimization technique [EN], Research Journal on Computer Engineering, K. Sankar, 2008 :

http://www.academia.edu/1144568/Train_Scheduling_using_Ant_Colony_Optimization_Technique

Après un rappel sur le comportement des fourmis et un état de l'art, l'application des fourmis au problème de planification des trains est présentée dans ce petit article.

Simulation en biologie, D. Leray, 2005 :

http://perso.limsi.fr/jps/enseignement/examsma/2005/3.simulation_2/LERAY/LERAY.HTML

Il s'agit de la page personnelle d'un chercheur, présentant une partie d'un de ses cours. On y trouve une présentation de la simulation en biologie, et l'application à un problème médical.

8. Réseaux de neurones

Classification par réseaux de neurones pour la reconnaissance de caractères. Application à la lecture de plaques d'immatriculation, C. Gratin, H. Burdin, O. Lezoray, G. Gauthier, 2011 :

http://hal.inria.fr/docs/00/80/90/27/PDF/AdcisISS2011.pdf

Les réseaux de neurones sont utilisés dans cet article pour lire des plaques d'immatriculation, qu'elles soient lisibles, mal éclairées ou abîmées. Les résultats sont présentés pour des plaques de différents pays.

An Application of Backpropagation Artificial Neural Network Method for Measuring The Severity of Osteoarthritis [EN], International Journal of Engineering & Technology, Vol. 11, no. 3, D. Pratiwi, D. Santika, B. Pardamean, 2011 :

http://arxiv.org/abs/1309.7522

La médecine utilise de nombreuses images, et les réseaux de neurones sont une technique très efficace pour trouver des patterns sur celles-ci. Dans cet article, les auteurs indiquent comment ils ont pu estimer la gravité d'arthrose à partir de radios de la main et du poignet.

Les réseaux de neurones avec Statistica, StatSoft, 2012 :

http://www.statsoft.fr/company/newsletter/17/
Classification_Scores_Credit_reseaux_neurones.pdf

Statistica est un logiciel de statistiques très connu. Ses créateurs envoient régulièrement des newsletters, et celle-ci est dédiée à l'utilisation des réseaux de neurones dans le logiciel.

Les réseaux de neurones, avenir du trading ?, NextFinance, J. Harscoët, 2010 :

http://www.next-finance.net/Les-reseaux-de-neurones-avenir-du

Dans cet article, l'utilité des réseaux en finance est discutée, en particulier pour l'estimation des valeurs boursières et l'optimisation des portefeuilles, dans un domaine très compétitif.

Exécution de trajectoire pour robot mobile par réseaux de neurones, The International Conference on Electronics & Oil : From Theory to Applications, G. Zidani, A. Louchene, A. Benmakhlouf, D. Djarah, 2013 :

http://manifest.univ-ouargla.dz/documents/Archive/Archive%20Faculte%20des%20Sciences%20et%20Technologies%20et%20des%20Sciences%20de%20le%20Matiere/The-INTERNATIONAL-CONFERENCE-ON-ELECTRONICS-OIL-FROM-THEORY-TO-APPLICATIONS2013/Gh_Zidani.pdf

Le contrôle de robot peut parfois être très complexe. Le temps de calcul du contrôleur peut donc être trop important pour être embarqué sur une application en temps réel. Le but de cet article est donc de reproduire un contrôleur existant, mais en obtenant un algorithme plus léger.

Annexe

1. Installation de SWI-Prolog

SWI-Prolog est un logiciel permettant d'utiliser Prolog sur les PC Windows et les Mac.
Le site officiel est : http://www.swi-prolog.org/

Le logiciel est disponible sur la page : http://www.swi-prolog.org/download/stable

La première étape consiste à télécharger le logiciel :

- Choisissez dans la section **Binaries** le programme adapté à votre machine.
- Enregistrez le fichier.
- Lancez l'installation en double-cliquant dessus.
- Acceptez l'invite de sécurité.

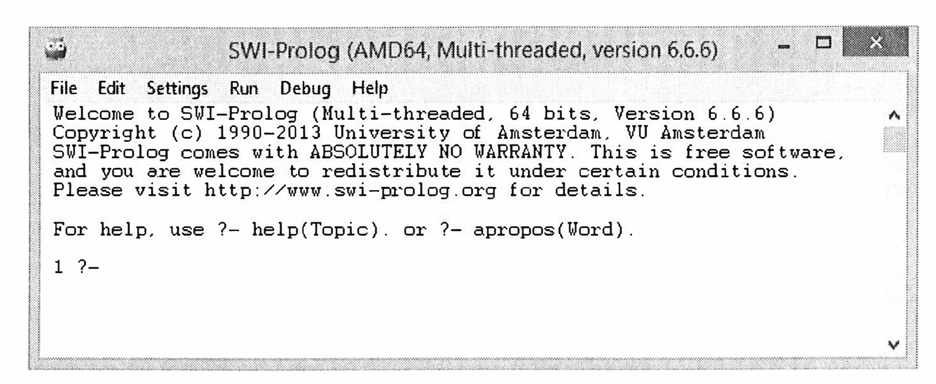
L'installation démarre alors. Les différentes étapes sont très simples :

- À la présentation de la licence, choisissez **I agree**.
- Choisissez ensuite une installation de type **Typical** (par défaut) et **Next**.
- Choisissez l'emplacement d'installation ou laissez le choix par défaut.
- Sur l'écran suivant, laissez les choix par défaut et choisissez **Install**.

Une fois l'installation terminée, il suffit de cliquer sur **finished** (et choisir ou non de lire le fichier Readme).

2. Utilisation de SWI-Prolog

Au lancement du logiciel, il se présente comme suit :



On y voit la console Prolog et le prompt, attendant une commande à exécuter.

Pour créer un nouveau projet :

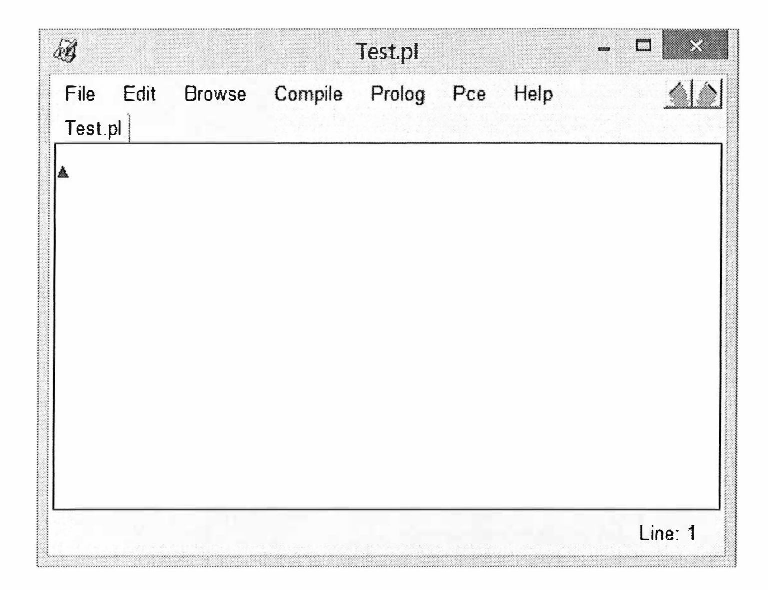
▶ **File** puis **New**.

▶ Choisissez l'emplacement du fichier à créer.

Remarque

Par convention, les fichiers prolog se terminent en .pl. Cependant, libre à vous de choisir l'extension voulue.

Le fichier (vide initialement) contenant les règles et prédicats s'ouvre dans une autre fenêtre :



C'est dans ce fichier qu'il faut écrire le contenu du moteur. Ici nous utilisons l'exemple du chapitre Systèmes experts :

```
manger(chat, souris).
manger(souris, fromage).

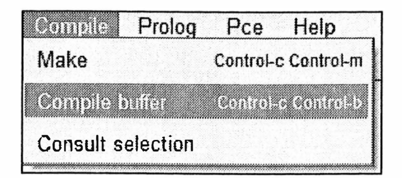
fourrure(chat).
fourrure(souris).

amis(X, Y) :-
     manger(X, Z),
     manger(Z, Y).
```

Pour pouvoir utiliser les prédicats et règles voulues :

- Écrivez l'exemple fourni ou votre propre contenu.
- Enregistrez le fichier.

Compilez-le : menu **Compile** puis **Compile buffer**.



Dans la console, une mention doit préciser que le fichier a été compilé. Sinon, corriger les erreurs et recommencer.

Une fois le code compilé, il est chargé dans la console et peut donc être utilisé. Il suffit alors d'indiquer ce que l'on souhaite dans cette dernière. Bien penser à terminer les lignes par un '.'. Pour obtenir les résultats suivants, appuyer sur ';'.

Voici un échange possible en console :

```
1 ?- fourrure(X).
X = chat ;
X = souris.

2 ?- manger(chat, X).
X = souris.

3 ?- amis(X, Y).
X = chat,
Y = fromage ;
false.

4 ?- fourrure(fromage).
false.

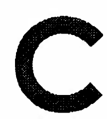
5 ?-
```

Remarque

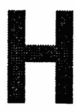
Attention, à chaque modification, bien penser à enregistrer le fichier et à le recompiler pour que celle-ci soit prise en compte dans la console.



B



D



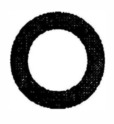
I

J

L

M

N



R

S




eni
Editions

L'Intelligence Artificielle pour les développeurs
Concepts et implémentations en C#

Ce livre sur l'**Intelligence Artificielle** s'adresse particulièrement aux **développeurs** et ne nécessite pas de connaissances mathématiques approfondies. Au fil des chapitres, l'auteur présente les **principales techniques d'Intelligence Artificielle** et, pour chacune d'elles, les inspirations, biologiques, physiques voire mathématiques, puis les différents concepts et principes (**sans entrer dans les détails mathématiques**), avec des **exemples et figures** pour chacun de ceux-ci. Les domaines d'application sont illustrés par des **applications réelles et actuelles**. Chaque chapitre contient **un exemple d'implémentation générique**, complété par une **application pratique, développée en C#**. Ces **exemples de code étant génériques**, ils sont facilement adaptables à de nombreuses applications C#, que ce soit en Silverlight, sur Windows Phone, pour Windows 8 ou pour des applications .Net plus classiques. Les techniques d'Intelligence Artificielle décrites sont :

- **Les systèmes experts**, permettant d'appliquer des règles pour prendre des décisions ou découvrir de nouvelles connaissances.
- **La logique floue**, permettant de contrôler des systèmes informatiques ou mécaniques de manière beaucoup plus souple que les programmes traditionnels.
- **Les algorithmes de recherche de chemin**, dont le A* très utilisé dans les jeux vidéo pour trouver les meilleurs itinéraires.
- **Les algorithmes génétiques**, utilisant la puissance de l'évolution pour apporter des solutions à des problèmes complexes.
- **Les principales métaheuristiques**, dont la **recherche tabou**, trouvant des optimums à des problèmes d'optimisation, avec ou sans contraintes.
- **Les systèmes multi-agents**, simulant des foules ou permettant des comportements émergents à partir de plusieurs agents très simples.
- **Les réseaux de neurones**, capables de découvrir et de reconnaître des modèles, dans des suites historiques, des images ou encore des données.

Pour aider le lecteur à **passer de la théorie à la pratique**, l'auteur propose en téléchargement sur le site www.editions-eni.fr, sept projets Visual Studio 2013 (un par technique d'Intelligence Artificielle), développés en C#. Chaque projet contient une **PCL, pour la partie générique**, et une application **WPF, pour la partie spécifique à l'application proposée**.

Le livre se termine par une bibliographie, permettant au lecteur de trouver plus d'informations sur ces différentes techniques, une sitographie listant quelques articles présentant des applications réelles, une annexe et un index.

Les chapitres du livre

collection DataPro

Virginie MATHIVET

Après un diplôme d'ingénieur INSA et un DEA «Documents, Images et Systèmes d'Informations Communicants», **Virginie MATHIVET** a fait une thèse de doctorat au sein du laboratoire LIRIS, en Intelligence Artificielle, plus précisément sur les algorithmes génétiques et les réseaux de neurones. Elle est aujourd'hui professeur permanent à l'EPSI de Lyon, où elle enseigne l'Intelligence Artificielle ainsi que des matières liées au développement (C#, PHP, Java, JS...), la modélisation 3D ou les méthodologies de développement. A travers ce livre, elle partage sa passion pour le domaine de l'Intelligence Artificielle et le met à la portée des développeurs pour qu'ils puissent en exploiter tout le potentiel.



sur **www.editions-eni.fr** :
→ Un projet Visual Studio par chapitre, contenant la PCL et l'application WPF illustrant l'exemple proposé dans le chapitre.

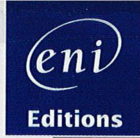
Pour plus d'informations :



45 €



isbn : 978-2-7460-9215-0



www.editions-eni.fr